**ANGULAR**



DE CERO A EXPERTO CREANDO APLICACIONES

|  |  |
| --- | --- |
| Resultado de imagen de udemy logo | CURSO UDEMY  Link: <https://www.udemy.com/angular-2-fernando-herrera/learn/lecture/6401730#overview>  **ÁLVARO MORÁN GARCÍA** |

# Introducción

## Sobre el curso

El curso está enfocado a la versión de Angular 8+. Las versiones de Angular están en constante cambio y hay un poco de lío de versiones. Aproximadamente cada 6 meses o incluso menos se actualiza la versión estable de Angular.

El Framework se llama Angular, sin coletilla. Se suele utilizar la coletilla de la versión del Framework de manera coloquial (Angular 2, Angular 4…). El curso está enfocado en Angular7, que será totalmente compatible con Angular8+; desde Angular2 se está intentando mantener la retrocompatibilidad entre las versiones.



## Aplicaciones necesarias

Se requieren las siguientes aplicaciones instaladas en el sistema para el desarrollo en Angular:

* **NodeJS**: Entorno de ejecución de JavaScript basado en el motor de JavaScript V8 de Chrome.
  + Link: <https://nodejs.org/es/>

Para comprobar que se ha instalado correctamente hay que verificar que tanto el NodeJS como su Package Manager (npm) su versión de instalación por consola:

|  |
| --- |
| node -v // Versión de NodeJS  npm -v // Versión del Package Manager |

* **Google Chrome:** No es obligatorio, pero es recomendado para analizar según qué código.
* **TypeScript:** Nos permitirá detectar errores de edición en nuestro código. Habiendo instalado con anterioridad NodeJS, podemos hacer uso de su Package Manager para instalar TypeScript, utilizando el siguiente comando en consola con permisos de administrador:

|  |
| --- |
| npm install -g typescript // Instalación de TypeScript  tsc --version // Versión del TypeScript |

* **Gestor de Angular:** Lo que nos permitirá crear las aplicaciones de Angular (también llamado **Angular CLI**). Se puede instalar utilizando el gestor de paquetes de NodeJS de la siguiente forma:

|  |
| --- |
| npm install -g @angular/cli // Instalación Angular CLI  ng --version // Versión del Angular CLI |

* **Ionic:** Framework que permite crear aplicaciones móviles basadas en Angular. Será necesario en algún momento del curso. Se puede instalar utilizando el gestor de paquetes de NodeJS:
  + npm install -g ionic

## Editores de texto

Se recomiendan dos alternativas:

* **Atom**: Algo pesado, pero bastante potente.
  + **Link**: <https://atom.io/>
* **Visual Studio Code**: De la suite de Microsoft Visual Studio, potente y ligero.
  + **Link**: <https://code.visualstudio.com/>

Se recomiendan instalar los siguientes plugin para facilitar el trabajo (es opcional):

* **Atom**:
  + Angular 2 Type Script Snippets
  + Atom Bootstrap3
  + Atom Typescript
  + File Icons
  + Platformio Ide Terminal
  + V Bootstrapt4
* **Visual Studio Code:**
  + Angular 2 TypeScript Emmet
  + Angular 5 Snippets – TypeScript, Html, Angular Material
  + Angular Language Service
  + Angular v5 Snippets
  + Angular2-inline
  + Bootstrap 4 & Font Awesome snippets
  + HTML CSS Support
  + JavaScript (ES6) code snippets
  + JS-CSS-HTML Formatter
  + JSHint
  + Material Icon Theme
  + Prettier – Code Formatter

A lo largo de este curso se recomienda el uso de Visual Studio Code:



# INTRODUCCIÓN A TYPESCRIPT Y ES6

## Introducción

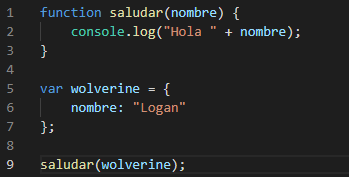
En este curso veremos una introducción a ambos lenguajes al lenguaje TypeScript y, especialmente, su estándar ECMAScript6. No son algo implícitamente ligados a Angular, pero es el lenguaje que se utilizará para generar los componentes en Angular mediante decoradores, añadirá un fuerte tipado de las variables, permitirá el uso de funciones de flecha, etc.

## TypeScript y compilación

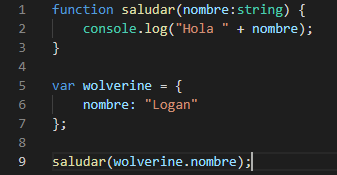
Todo el código que generemos como TypeScript será traducido a JavaScript, existiendo una fuerte relación entre ambos lenguajes. TypeScript es tan solo una capa adicional a JavaScript que permite dotarle de una funcionalidad extra; no obstante, en el momento de realizar la compilación de proyectos en TypeScript, los archivos generados son archivos JavaScript comunes.

TypeScript nos proporciona una serie de beneficios a la hora de gestionar tipos de datos. Por ejemplo, veamos la siguiente diferencia de código:

* **JavaScript:**



* **TypeScript**:



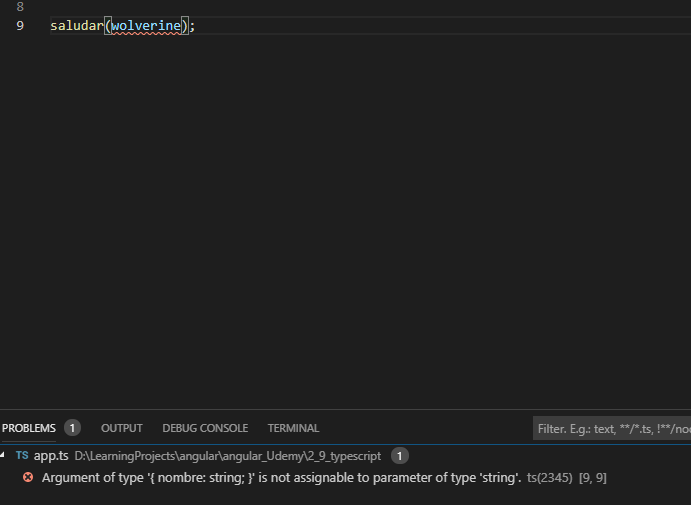
En TypeScript podemos especificar el **tipo** de argumento de la función. Adicionalmente, cuando definimos la variable wolverine, que puede tener muchas otras variables internamente y ser tan complejo como deseamos, es necesario especificar cuál de esas variables queremos utilizar. El lenguaje con TypeScript es más fuertemente tipado que en JavaScript, como vemos.

Una vez generado el código en TypeScript se compila mediante el comando en consola:

|  |
| --- |
| tsc app.ts // Compilación aplicación app |

Si nuestro código es válido este proceso generará un archivo de JavaScript (extensión js) totalmente válido.

Si hay algún problema con el código TypeScript que se genera, en el editor podemos ver los errores de compilación resaltados, como sucede con lenguajes de programación como Java, donde los errores en tiempo de compilación son mostrados en el mismo IDE:



## Configuración de TypeScript

Dependiendo del IDE que estemos utilizando y de cómo hayamos generado el proyecto de TypeScript, es posible que necesitemos generar un archivo que define los parámetros de la compilación del código. Para ello se debe ejecutar el siguiente comando por consola:

|  |
| --- |
| tsc -init // Configura la compilación TypeScript |

Esto crea un archivo json con configuración sobre la compilación de nuestros archivos TS:



Si, posteriormente, ejecutamos:

* tsc

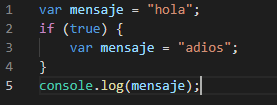
se buscarán todos los cambios y se ejecutará el compilador en base a lo configurado en el json. A partir de este momento el compilador estará monitorizando permanentemente todos los cambios que se produzcan en el código y no será necesario estar pendiente de volver a escribir los comandos en la consola.

En cualquier caso, y aunque esto de momento solo sea necesario en Atom, es necesario saber que las características de compilación se definen en ese json que se genera de la forma especificada con anterioridad, pues en el futuro podemos querer modificar dichas características de compilación.

Sí que es útil en Visual Studio también para que tsc esté recompilando todo automáticamente todo el código una vez que es salvado. Es posible que haga falta algún plugin adicional para ello, como el TypeScript Auto Compiler.

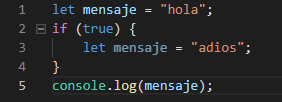
## Variables let y const

En versiones 5 y anteriores de JavaScript se solían declarar las variables tal que así:

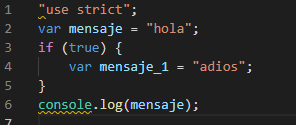


Si ejecutamos este código lo que veremos en el navegador web será adios, lo cual es lógico porque dentro del condicional se está cambiando el valor inicial del mensaje.

Si simplemente cambiamos var por let:

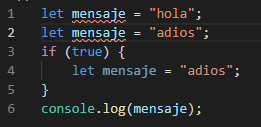


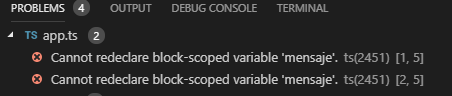
La salida ahora del programa será hola. Esto se debe a que cuando llamamos a la consola para que escriba el mensaje, toma la variable mensaje que está en su mismo scope, que es el hola. La redefinición de mensaje que hemos realizado solo aplica al scope dentro del if. Si comprobamos como es el código js generado al compilar el ts, vemos lo siguiente:



El compilador ha detectado automáticamente que había dos declaraciones de variables iguales y ha cambiado el nombre de la que está en el scope del if para evitar sobreescribir a la otra. Esta funcionalidad es bastante útil cuando tenemos un código bastante extenso y queremos tener controladas los valores de nuestras variables.

Si definimos en el mismo scope dos variables que se llamen igual, el mismo IDE nos impide hacerlo:

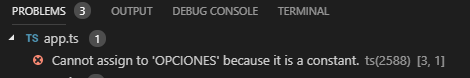




Por ello a partir de ahora es más recomendado el uso de let en vez de var.

Para definir constantes, por otro lado, se utiliza la palabra reservada const (seguido del nombre de la variable en mayúsculas, por convención). Si utilizamos una constante y luego intentamos cambiar su valor, el propio IDE nos impedirá hacerlo.



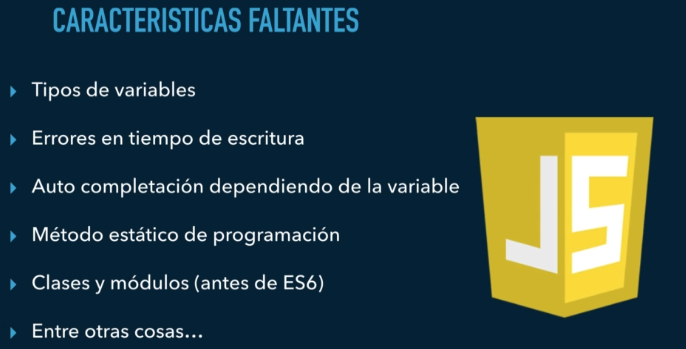


Al igual que sucede con let, si podemos definir constantes que se llamen iguales en distintos scopes. El compilador de TypeScript se encargará de resolver esa incidencia de forma similar a lo que hemos visto anteriormente con mensaje\_1.

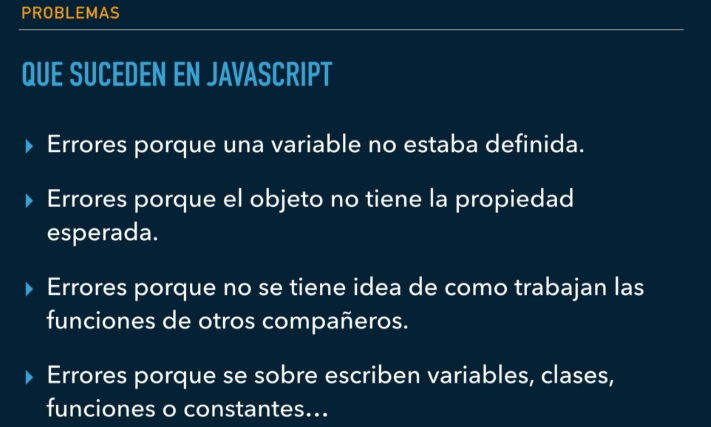
Estas todas son funcionalidades propias de ECMAScript6 que se irán utilizando a lo largo del curso.

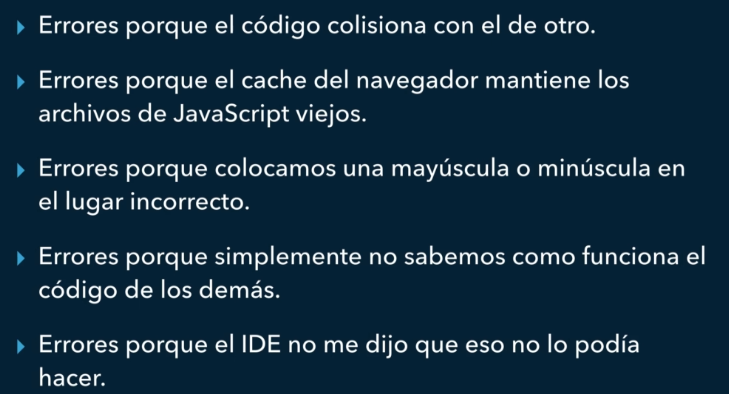
## Introducción al TypeScript

Hay una serie de características de las que adolece JavaScript y que TypeScript intenta subsanar. Son las siguientes:



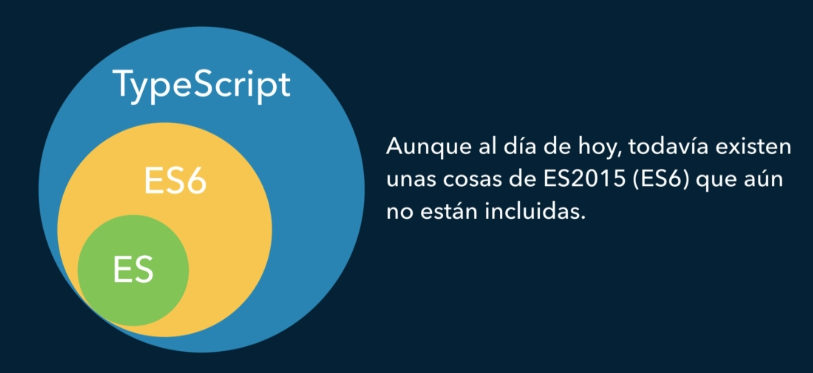
Así mismo hay una serie de errores que suceden en JavaScript, que también intentarán ser subsanadas por TypeScript:

****

****

El problema de todos estos errores es que en JavaScript solo los puedes detectar una vez que el código está compilado. TypeScript pone a JavaScript al nivel de otros lenguajes de programación como Java o C#, donde el IDE es capaz de realizar una pre-compilación y saber si va a haber problemas de manera previa a la compilación.

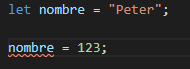
TypeScript fue creado por Microsoft y no deja de ser un super-set de JavaScript que proporciona soluciones a los problemas mencionados anteriormente:

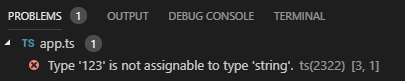


## Tipos de Datos

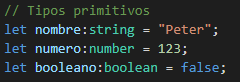
Veamos como manejamos los tipos de datos en TypeScript.

Si inicialmente creamos una variable con let de un tipo texto y luego le cambiamos el tipado a entero, TypeScript se nos quejará:

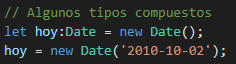




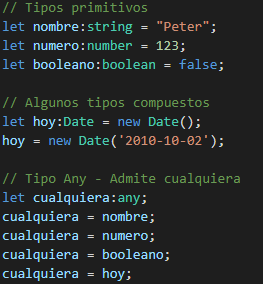
Una forma de declarar los tipados explícitamente y que sea más visible a la hora de programar es especificando el tipo en el código (en este ejemplo veremos un set de los tipos primitivos más utilizados):



También hay algunos tipos compuestos, como puede ser Date para almacenar fechas (muy similar a java, llamando a un constructor con y sin argumentos en este ejemplo):



Es muy interesante el tipo any, que puede guardar cualquier tipo de dato. Nos proporciona la flexibilidad de JavaScript (esa ligereza a la hora de tratar los tipos) pero desde TypeScript:

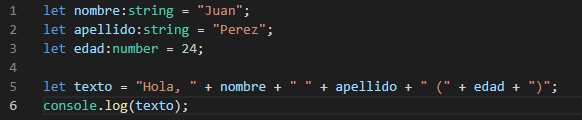


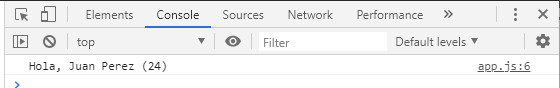
Podemos definir tipos de manera compuestas, aunque será necesario aclarar que luego siempre que queramos redefinir el tipo deberemos de volver a especificar todos sus atributos:



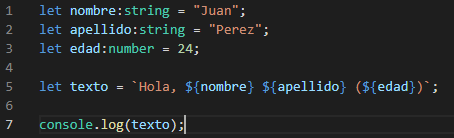
## Templates literales en ES6

Las plantillas literales son como un string que tienen una serie de características especiales. Supongamos por ejemplo que queremos representar la información de una persona en el log, donde vemos que cuanta más información literal queremos representar más farragoso se vuelve el código:

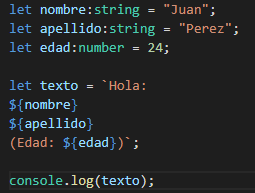




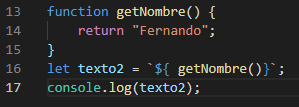
ES6 nos proporciona una serie de utilidades para manejar mejor estas cadenas de literales. Si utilizamos el backtick (apóstrofes inclinados) nos permitirá meter variables dentro de la cadena mediante el uso de $ y {}:



El backtick nos permite además crear scripts de multilínea sin necesidad de introducir saltos de línea, simplemente traduce todo el template literal con saltos de línea, tabulaciones y lo que sea dentro de un string.

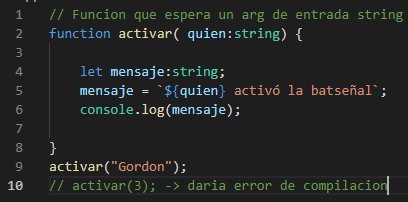


Es importante saber que todo el código que se sitúe dentro de los caracteres ${ } es código javascript puro. Por eso al meter el nombre de la variable simplemente nos la escribe, pero podemos meter cualquier otra cosa que sea código javascript, tan complejo como se quiera:

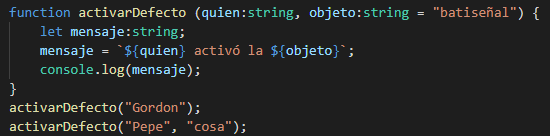


## Funciones

Las funciones son secciones de código con su propio scope, que pueden tener o no argumentos de entrada. Si los argumentos de entrada son obligatorios, la declaración del método y su acceso es tal que así:

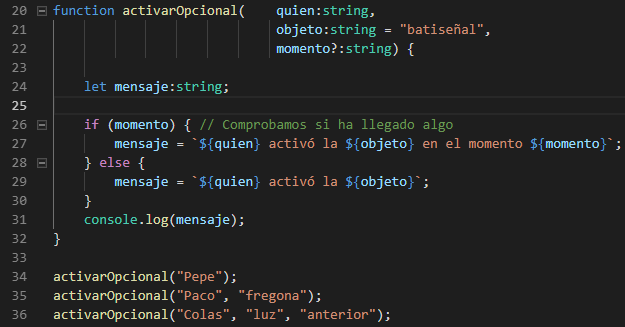


Pero también contamos con argumentos de entrada que son opcionales. En ese caso su uso es como el que sigue:



En el caso anterior lo que sucede es que, si no estamos especificando el argumento de entrada objeto, se utilizará un valor por defecto.

Adicionalmente, podemos establecer argumentos de entrada opcionales de verdad (sin valor por defecto) del siguiente modo:

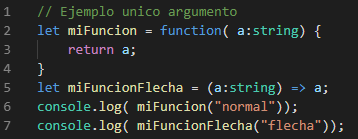


Cuando utilicemos este tipo de argumentos de entrada debemos de tener en cuenta desde el código que puede que la variable opcional no haya sido inicializada y tener cuidado de no utilizarla en ese caso. Para detectar si algo ha sido o no inicializado, podemos simplemente meterlo dentro de una cláusula if (en caso de ser nulo o no haber sido inicializado, se considerará un resultado false para esa comprobación).

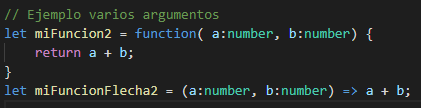
**Nota: De la misma forma podemos definir objetos o componentes que tengan argumentos opcionales, mediante el carácter ¿**

## Funciones de flecha

Comencemos primero por ver la distinta sintaxis entre ambas funciones que realicen la misma función, en este caso, devolver el valor pasado por argumento (especificando los tipados de los argumentos para que no se nos queje TypeScript):



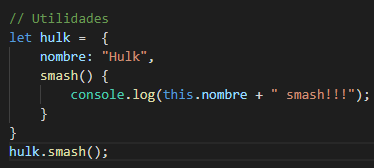
Con la sintaxis de las funciones comunes estamos ya habituados. La función flecha directamente, definimos los argumentos de entrada después del igual y utilizamos la flecha para mostrar que es lo que se devuelve. Con varios argumentos se haría como sigue:



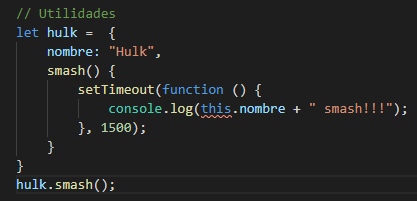
Si el cuerpo de la función es más complejo, esto es, requiere más líneas de código, se hará de la siguiente forma, siendo necesario especificar en qué línea se realiza la devolución del resultado:



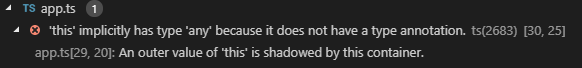
¿Cuál es su utilidad? Supongamos el siguiente escenario: podemos crear una clase que tenga una clase propia que cuente con una serie de métodos, por ejemplo:



Esto sigue siendo posible sin la función flecha. Pero supongamos que el método es un poco más complejo y tiene que llamar a otro método dentro, por ejemplo, el siguiente escenario:

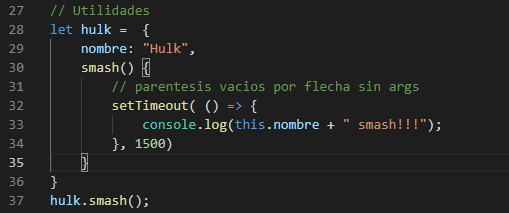


Esto nos devuelve el siguiente error cuando intentamos acceder al nombre:



El this dentro del timetout no apuntaría a dentro de nuestro objeto hulk, sino que apunta a nuestro objeto padre de todo, fuera del scope de nuestra variable creada. Perdemos el scope de nombre de hulk y el this apunta al scope global del archivo.

La función flecha evita que esto suceda. Se puede evitar implementando lo siguiente:

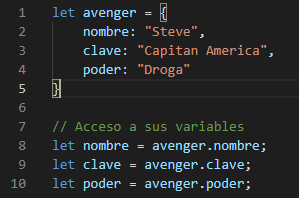


En este caso no da error de compilación y el scope de nombre dentro de nuestra variable que nos hemos creado es accedido correctamente, incluso llamándolo desde un método dentro de otro método.

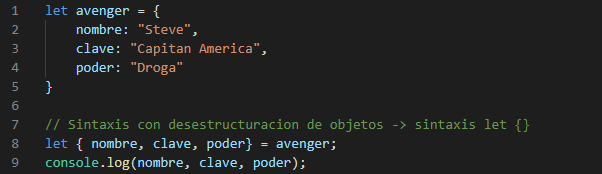
## Desestructuración de objetos y arrays

### Desestructuración

Supongamos el siguiente ejemplo, donde nos creamos un objeto determinado con sus variables y accedemos a ellas:



Con la desestructuración de objetos podemos acceder a esas variables de una forma más cómoda, siguiendo el siguiente ejemplo:



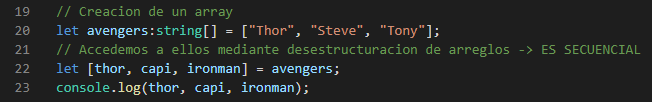
De esta forma creamos automáticamente los tipos que tiene dentro la clase avenger que hemos creado. En este caso es importante hacer notar que dentro de la sintaxis let {}, **los dos puntos ( : ) no funcionan para indicar tipos de datos, si no que funcionan como alias**:



A partir de este momento, el poder del avenger no será poder, sino alias2, por ejemplo. TypeScript ya sabe el tipo que tienen los datos en el momento que los declaramos dentro del objeto avenger y es por ello por lo que no es necesario (y no se permite) el establecer los tipos mediante desestructuración, simplemente serán alias para facilitar el uso de estos datos en otro contexto.

### Arrays

Respecto a los arrays, los declaramos de la siguiente forma para declararlos y poder acceder a ellos:



Es importante tener en cuenta que la desestructuración cuando accedemos a arrays es secuencial, esto es, el alias que asignamos primero se corresponderá con la primera posición del array y así sucesivamente.

Los arrays pueden ser declarados, además, de otra forma que puede ser considerada como más clara (o más similar a otros lenguajes como Java):

|  |
| --- |
| let fruits: Array<string>;  fruits = ['Apple', 'Orange', 'Banana'];  let ids: Array<number>;  ids = [23, 34, 100, 124, 44]; |

Por supuesto, se pueden crear arrays sin especificar un tipo concreto y metiendo cualquier tipo de objeto dentro, pero perdemos las funcionalidades propias de TypeScript y su tipado fuerte:

|  |
| --- |
| let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false]; |

Existe una forma intermedia de abordar el tipado en listas; TypeScript nos permite especificar listas con varios tipos concretos, y no solo uno. Esto se hace de la siguiente forma:

|  |
| --- |
| let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];  // or  let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana']; |

Para acceder a los elementos del Array podemos utilizar tanto los índices como acceder utilizando bucles o cualquier otro método:

* Mediante el uso de índices:

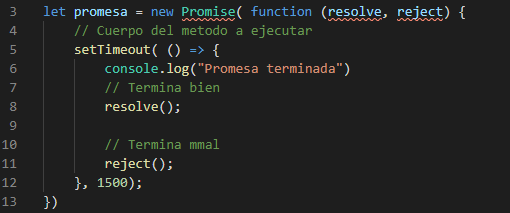
|  |
| --- |
| let fruits: string[] = ['Apple', 'Orange', 'Banana'];  fruits[0]; // returns Apple  fruits[1]; // returns Orange  fruits[2]; // returns Banana  fruits[3]; // returns undefined |

* Mediante el uso de bucles:

|  |
| --- |
| let fruits: string[] = ['Apple', 'Orange', 'Banana'];  for(var index in fruits)  {  console.log(fruits[index]); // output: Apple Orange Banana  }  for(var i = 0; i < fruits.length; i++)  {  console.log(fruits[i]); // output: Apple Orange Banana  } |

## Promesas

Las promesas son una utilidad de ES6 para suplir carencias de JavaScript. Sirve para ejecutar una determinada tarea cuando una tarea síncrona es terminada. La sintaxis a seguir es la siguiente:



Para crear una promesa es necesario pasarle una función con dos argumentos, que por convención se deben llamar resolve y reject. Resolve será lo que sucede cuando la función termina correctamente mientras que reject será lo que el proceso asíncrono se resuelva de forma incorrecta. Para llamar a la función hacemos lo siguiente:

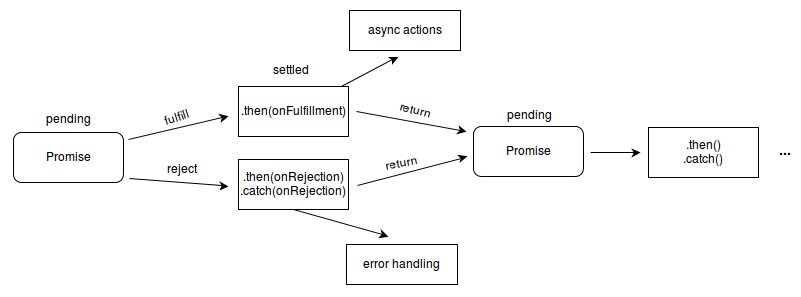


Cuando llamamos a la función de promesa tenemos que especificarle que queremos que haga en cada uno de los casos que hemos definido, esto es, que queremos que haga cuando se ejecute todo correctamente o cuando haya algún error (de ahí las dos function que pasamos como argumentos que se corresponden con resolve y reject).

Al final se trata de que métodos asíncronos se comporten como si fuesen síncronos; en vez de retornar inmediatamente el valor final, se retorna la promesa de que se suministrará un valor una vez haya terminado su procesamiento. Una promesa se encuentra siempre en uno de los siguientes tres estados:

* Pendiente (pending): Su estado inicial, antes de que termine de evaluarse completamente.
* Cumplida (fullfilled): La operación se ha realizado de forma satisfactoria.
* Rechazada (rejected): La operación ha fallado.

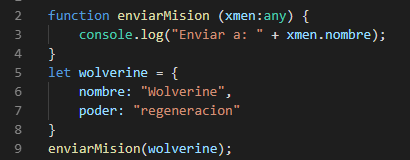
Las promesas pueden ser cumplidas con un valor o bien rechazadas con una razón. Una vez que se ha obtenido uno de estos dos posibles resultados, la promesa llama a los métodos que se le han pasado como argumentos para gestionar cada uno de los resultados.



## Interfaces de TypeScript

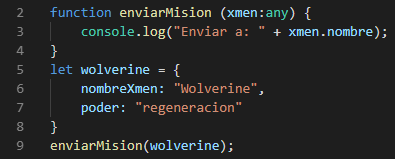
Las interfaces en TypeScript se asemejan a lo que sucede con casi todos los lenguajes POO; definen un set de comportamientos esperados que cualquier clase que implemente dicha interfaz debe cumplir. Esto nos permite manejar clases de distintas naturalezas pero que implementen una misma interfaz en base solo a esa interfaz, sin necesidad de entrar en las particularidades de cada clase.

Supongamos que intentamos realizar lo siguiente en JavaScript:

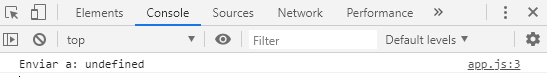


Esto funciona sin mayor problema, la función enviarMision espera cualquier objeto de entrada que tenga un nombre al que pueda acceder para poder imprimirlo por consola.

Sin embargo, si cuando creamos el objeto la variable interna no se llama exactamente nombre, sucede lo siguiente:



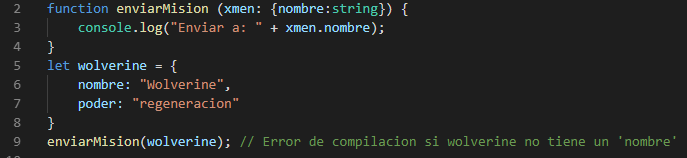
Wolverine no tiene ningún atributo nombre, sino nombreXmen. Y el compilador no nos da ningún tipo de error. Si lo ejecutamos y vemos el resultado en el explorador web, vemos lo siguiente:



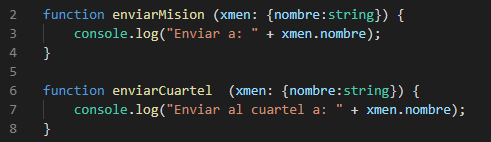
Lo cual es esperado, no puede acceder a la variable nombre porque no existe dentro de wolverine.

Esto se puede resolver haciendo que enviarMision espere objetos determinados o que cumplan determinadas características. Lo podemos hacer de dos formas posibles:

* Especificando que queremos una variable que se llama nombre con su tipo de dato. Es una solución poco elegante pero que puede servir:

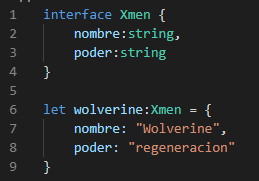


Esta aproximación tiene problemas obvios en cuanto tratamos con objetos muy grandes que tenga muchos nombres o propiedades. Esto se ve agravado todavía más si tenemos varios métodos que son llamados por este tipo de objetos:

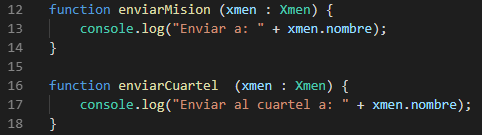


Donde tendríamos que gestionar los argumentos de entrada de todos los métodos para adecuarse al objeto que queremos enviarles. Es una forma de programar muy descentralizada y que nos dará quebraderos de cabeza.

* Creando interfaces, que servirán para definir determinados comportamientos. Definimos una interfaz que va a cumplir nuestro objeto wolverine de la siguiente forma:



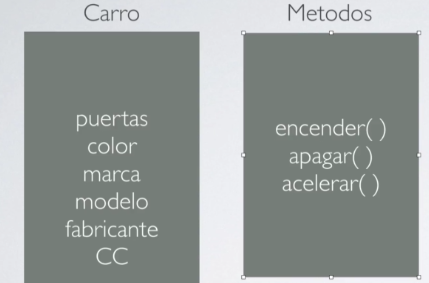
Y ahora en cada uno de los métodos no nos tenemos que preocupar de qué tipo de objeto está accediendo a ellos, simplemente nos tenemos que preocupar de que cumplan la interfaz Xmen para acceder a su nombre:



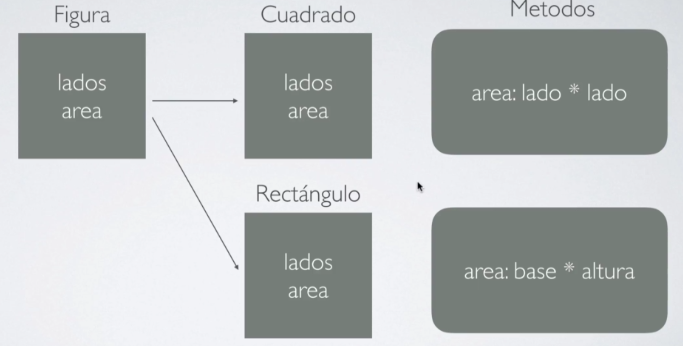
Nuestro objeto wolverine tendrá que tener desde este momento un atributo nombre porque de no ser así no cumplirá la interfaz y dará un error de compilación, pero solo en el sitio en que estamos declarando a wolverine de forma incorrecta.

## Introducción a clases (Conceptos generales de POO)

Una clase es un tipo de objeto determinado que tiene una serie de propiedades o características que describen dicha clase. Cuenta además con una serie de funcionalidades particulares (funciones o métodos) que realiza típicamente con las características que describen dicha clase. Veamos el siguiente ejemplo con un coche:



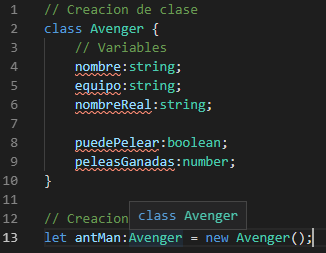
Además de ello, en POO es interesante las relaciones entre objetos, específicamente las relaciones de herencia. Podemos suponer una clase Figura, que es la clase padre de Cuadrado y Rectángulo, que son otras clases distintas pero que heredan de figura, son hijas de ella, y heredan también todas sus funcionalidades. No obstante, la implementación de estas funcionalidades cambia en función del tipo de la clase hija (aunque a todas las figuras se pueda calcular el área, a un cuadrado no se le calcula el área de igual forma que un rectángulo, por ejemplo):



Estos conceptos de herencia, clases y objetos que son propios de POO aplicarán también a la gestión de clases que nos proporciona TypeScript.

## Definición de una clase en TypeScript

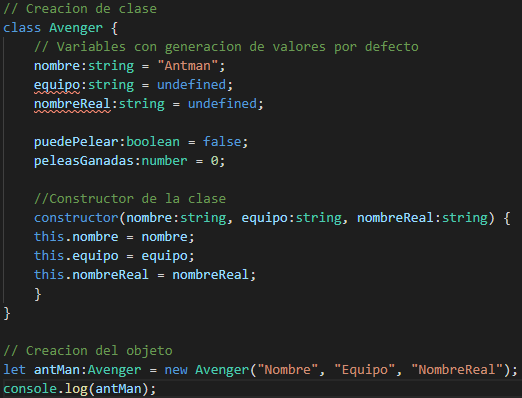
Las clases se definen especificando sus atributos de la siguiente forma:



Para crear un objeto de la clase, se llama a new y el nombre de la clase, de igual forma a como se hace en Java. En este caso no hemos especificado un constructor así que la llamada se produce al constructor por defecto sin argumentos.

## Constructores de clase en TypeScript

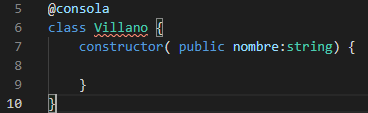
Al igual que en otros lenguajes con objetos, podemos definir constructores para inicializar variables con unos valores que nosotros deseemos o implementar en estos constructores algún tipo de código a ejecutar antes de que nuestra clase esté disponible para ser accedida por otras secciones del código.



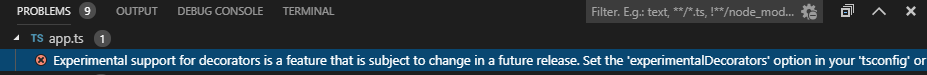
En este caso definimos unos valores por defecto para los atributos de la clase y en el constructor permitimos que algunos de esos parámetros (nombre, equipo, nombre real) sean inicializados a otro valor. Ahora cuando llamamos al constructor de la clase será necesario pasarle todos esos argumentos de entrada.

## Decoradores de clases

Los decoradores son una característica todavía experimental en TypeScript. Funcionan de manera similar a los decoradores en Java. Supongamos el siguiente ejemplo con un decorador (y un constructor simplificado, que equivale a los constructores que hemos visto anteriormente, pero simplificada la asignación de valor a variable de clase en una única línea):



Aunque este código se puede ejecutar, nos sale el siguiente aviso en nuestro terminal:



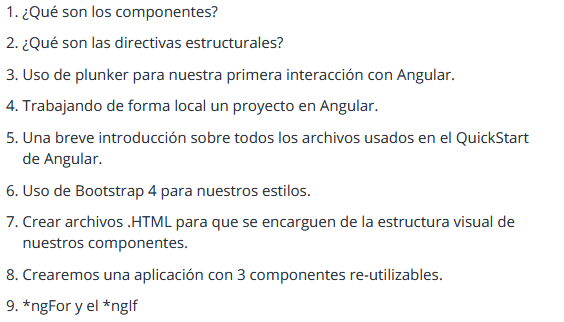
Podemos actualizar la configuración de nuestro compilador en el archivo json que se genera siempre que hacemos tsc -init para incluir esta característica experimental y que no nos genere estos avisos de compilación:



# Hola Mundo

## Introducción

En esta sección aprenderemos el funcionamiento básico de Angular, respondiendo entre otras cosas a las siguientes cuestiones:



## Componentes y directivas estructurales

Una aplicación en Angular está construida en base a distintos componentes. Entendemos como componentes distintas secciones de la aplicación diferenciadas como pueden ser:

* Menu de negación
* Barra lateral
* Pie de página
* Rerepresentación de página



Los componentes son clases normales que tendrán unos decoradores particulares que iremos viendo.

Por otro lado, las directivas estructurales son como instrucciones que le dicen a la parte de la representación HTML qué tiene que hacer. Tenemos por ejemplo las siguientes directivas:

* ngIf: Se encarga de mostrar u ocultar elementos HTML en la página web. Más que ocultar, deja de existir en el código HTML.
* ngFor: Realiza repeticiones de elementos HTML en una página. Mediante variables podemos cambiar la información que se representa en cada uno de esos elementos.

## Primera interacción en Angular

La página principal de Angular se tiene que convertir en nuestra primera fuente de referencia de información sobre Angular, sus elementos. Recordamos que el link para acceder a ello es el siguiente:

<https://angular.io/>

Hay una serie de páginas muy útiles para hacer páginas de demostración de Angular, obtener información al respecto, ver implementaciones y demás, que son las siguientes:

* <http://plnkr.co/plunks>
* <https://stackblitz.com/>

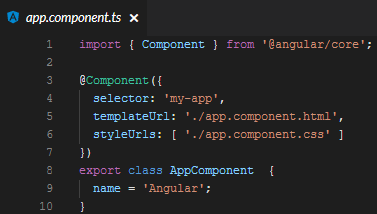
Veamos cómo sería la estructura básica de una aplicación Angular creada de 0 en este caso con StackBlitz:

* Contenido de index.html: Como siempre en HTML, es la primera página que se llama cuando se ingresa en un dominio. Su contenido es como sigue:



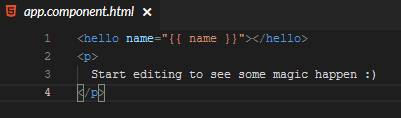
El loading es el mensaje que se muestra por defecto cuando está cargando la aplicación. Una vez que termina se sustituye por el contenido de la aplicación, esto es, el contenido de my-app. Se puede dejar vacío, por tanto, ya que el texto será inmediatamente sustituido al cargar la página web.

* app.component.ts: Contenido de la aplicación principal, que se llamado desde el index.html.



Cuando se encuentra en HTML un tag con el nombre my-app (o cualquier otro que especifiquemos en el campo selector) el elemento se renderizará en base a lo que se haya especificado en dicho componente (ver decorador component). Se trata de una clase de TypeScript normal (AppComponent). El decorador Component nos permite utilizarlo en el HTML, con la siguiente configuración:

* + **selector**: Va a ser el nombre que se utilizará desde el HTML
  + **templateUrl**: Será lo que se renderice gráficamente cuando utilicemos este elemento.
  + **styleUrls**: La hoja de estilos, con el estilo gráfico que se aplicará a este elemento cuando se renderice.
* App.component.html: Lo que antes se ha referenciado desde el templateUrl. Tiene el siguiente contenido:



Las referencias entre {{ }} son referencias a variables o propiedades propias de la clase generada en el TypeScript. Esto es, en este caso, hace referencia a Angular:



El resto es código HTML común (podemos sustituir hello por h1, como en HTML para indicar que es cabecera).

## Creación de un entorno local de Angular

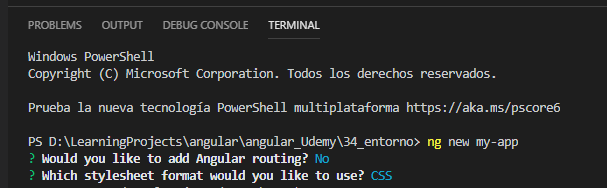
Para poder crear un entorno local de desarrollo de Angular se puede seguir la guía que está en el siguiente enlace:

<https://angular.io/start>

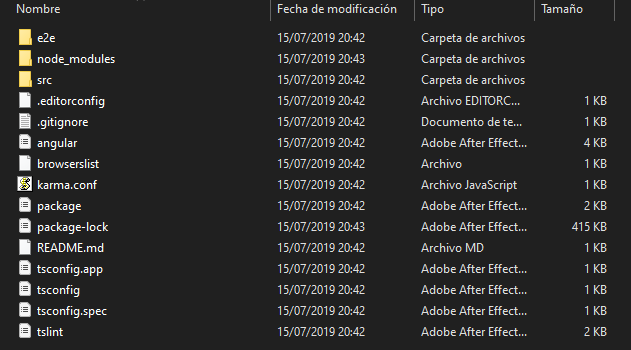
Es importante tener en cuenta que se aconseja utilizar el prompt por comandos Angular CLI, que ahorra bastante tiempo a la hora de generar una aplicación en Angular. Ya deberíamos tener instalado todo lo necesario para ello (sección 1).

Mediante esta línea de comandos simplemente nos tenemos que posicionar donde queremos crear el proyecto de Angular y ejecutar el comando siguiente:

|  |
| --- |
| ng new my-app |



Siendo my-app el nombre de la aplicación. Esto se puede ejecutar directamente desde la consola del Visual Studio Code. El resultado debería ser todo un entorno de desarrollo Angular creado:

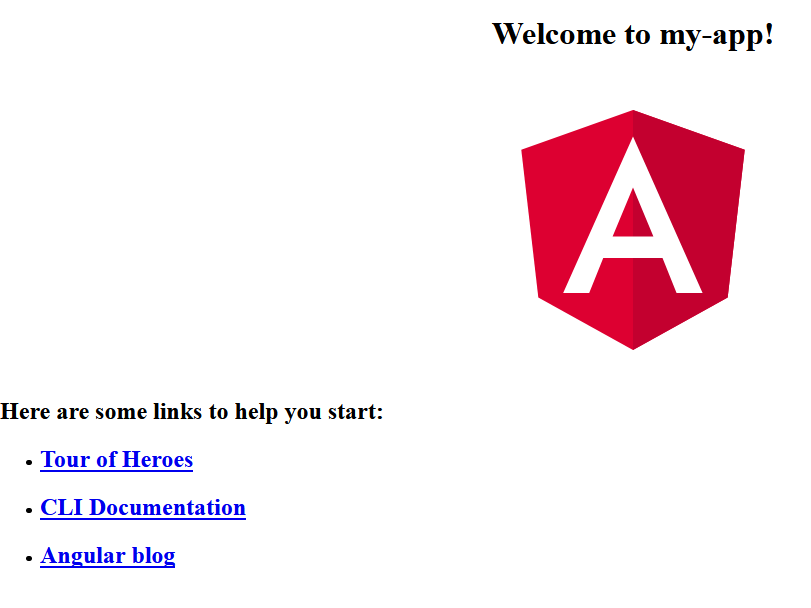


Podemos cambiar el nombre del proyecto simplemente cambiando el nombre de la carpeta de my-app a cualquier otra cosa, por ejemplo, 01-hola-mundo (es mejor evitar poner espacios en el nombre ni caracteres raros).

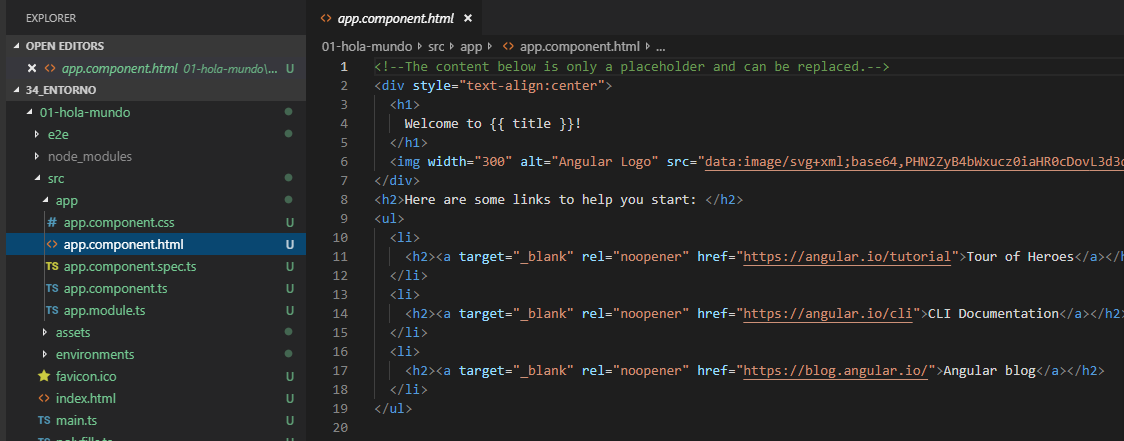
Para ejecutar la aplicación deberemos ejecutar el siguiente comando en la consola (por defecto usará el puerto 4200, podemos utilizar cualquier otro puerto mediante el flag -p XXXX):

|  |
| --- |
| ng serve  ng -o serve // abrirá el explorador web directamente |

Si añadimos el flag -o automáticamente se abrirá el explorador predeterminado para visualizar la aplicación corriendo:

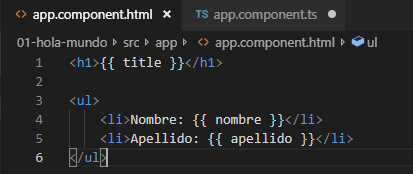


Vamos a analizar un poco el contenido de todo un proyecto de Angular. Para ello primeramente abrimos la carpeta app y buscamos nuestro app.component.html:

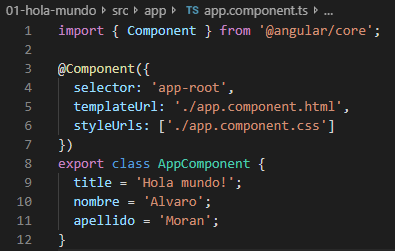


Su contenido es exactamente el HTML que se está mostrando en el explorador web. Si modificamos se debería de regenerar el proyecto y actualizar la representación visual en el explorador web automáticamente. De no ser así, podemos volver a ejecutar el comando que ejecutamos con anterioridad para levantar el servidor. Es importante tener en cuenta que para que esto funcione correctamente se debe ejecutar el comando en consola (o el Visual Studio Code entero, a tal efecto) en modo administrador.

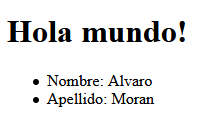
Vamos a modificar el app.component.html para mostrar algo que nosotros queramos, por ejemplo, nuestro nombre y apellidos. El contenido será el siguiente:



Tenemos que modificar nuestro archivo de TypeScript para que nuestro componente tenga las variables que estamos referenciando entre los {{ }} de la siguiente forma:



Con lo que el resultado gráficamente es el siguiente:



Desde consola podemos abrir el proyecto si escribimos el siguiente comando:

|  |
| --- |
| ng serve  ng -o serve // abrirá el explorador web directamente |

Una vez que tenemos el proyecto creado y está abierto en el explorador web, si todo va correctamente el proyecto se recompilará automáticamente al detectar cambios y estos serán inmediatamente visibles en el explorador (posiblemente siendo necesario un refresco de la página mediante F5).

## Estructura del proyecto

En este apartado analizaremos el contenido del proyecto carpeta a carpeta y nos familiarizaremos con aquellos archivos más importantes:

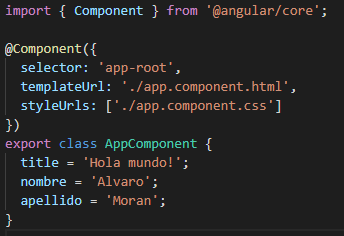
* **Carpeta e2e**: End to end. Se utiliza en general para hacer pruebas unitarias sobre el código. No se verá en este curso.
* **Carpeta node\_modules:** Todos los paquetes que se instalan cuando generamos el proyecto Angular. Aquí hay todo lo necesario para montar el servidor, escuchar los cambios etc. Son utilidades con fin de desarrollo. Una vez que la aplicación está finalizada se corre un script que corre sobre esta carpeta para cargarse todos aquellos módulos que no sean necesarios. Es desaconsejable subir esta carpeta a los repositorios antes de realizar esta operación, ya que su tamaño inicial con todas las herramientas de desarrollo es muy elevado. De normal esta carpeta es autoexcluida en el gitignore si se ha detectado que se ha creado sobre una carpeta que está siendo sincronizada con un repositorio. De no ser así y no haberse generado el archivo con la excepción, se puede operar de la siguiente manera:
  + Borrar toda la carpeta
  + Subir el proyecto al repositorio
  + Cuando se quiera descargar el proyecto, una vez descargado ejecutar sobre él el comando npm install, que volverá a generar la carpeta.

Esta carpeta se genera siempre en base a lo que indica el archivo package.json, que lo veremos más adelante.

* **Carpeta src:** Donde se guarda nuestra aplicación de Angular. Dentro de esta carpeta nos encontramos los siguientes archivos relevantes:
  + app: El contenido de la aplicación Angular, sus componentes y demás.
  + index.html: Es una página web común y corriente, prácticamente HTML plano salvo por la referencia a nuestra aplicación en Angular, que es donde se renderizará todo el contenido de nuestra aplicación Angular:



* + app.component.ts: Es donde se define el componente principal de nuestra aplicación Angular (el app-root al que dirige el index.html):



* + app.component.css: Es el estilo que se aplica a un componente, en este caso, app.component (ver la referencia templateUrl en la imagen anterior), nuestro componente principal. Cuando tengamos más componentes cada uno de ellos podrá tener su estilo particular.
  + app.component.html: El código HTML asociado al componente, lo que se renderizará finalmente de manera gráfica.
  + app.component.spec.ts: Los archivos que tienen el nombre de spec se refiere a archivos de pruebas automáticas. No se tocará en este curso, forma parte de un nivel de Angular más avanzado.
  + app.module.ts: Es una clase de tipo NgModule (definido por su decorador). Se analizará más adelante su comportamiento.
  + Carpeta assets: Recursos estáticos a utilizar por la aplicación, como imágenes, sonidos y demás.
  + Carpeta environments: Tiene dos archivos, para definir las variables de entorno en desarrollo y en producción (prod.ts). Más adelante se modificarán estas variables de entorno en el curso.
  + browsertlist: No es necesario, es propio de nuevas versiones de Angular y sirve para una mejor interpretación y soporte de los CSS.
  + karma.conf.js: Archivo de configuración de las pruebas de Karma. No se tocará en este curso.
  + main.ts: Es el primer código que Angular va a lanzar, al igual que cualquier main de otro lenguaje como Java. Por defecto nos dice que se ejecutará en un browser nuestra aplicación.
  + polyfills.ts: Ayudan a la compatibilidad entre versiones viejas de navegadores web. No se tocará manualmente.
  + styles.css: Estilos globales a la aplicación.
  + tests.ts: Archivo de tests que no se verá en este curso.
  + Tsconfig.app.json: Configuraciones de TypeScript. No se tocará en este curso.
  + Tsconfig.spec.json: Configuraciones de tests para la aplicación. No se tocará en este curso.
  + tslint.json: Configuración de cómo se representarán los errores cuando estamos escribiendo código. No se tocará en este curso.
* .editorconfig: Configuraciones para el editor. No es muy importante y rara vez se tocará.
* .gitignore: Archivo propio de git que le dice que cosas se subirán y qué cosas no se subirán al repositorio. Por defecto está configurado para que la carpeta node\_modules, entre otras, no se suban al repo.
* angular.json: Archivo que dice cómo es nuestra aplicación y cómo funciona. Define cosas importantes como los assets que utilizaremos, los estilos que utilizaremos y los scripts. Son cosas que se utilizarán más adelante y pocas veces tendremos que tocarlo manualmente.
* package-lock.json: Es como un log de como se ha creado el archivo package.json. Se genera automáticamente y no necesitamos hacer nada sobre él, es meramente informativo.
* package.json: Uno de los archivos más importantes, donde se establecen dependencias de nuestra aplicación entre otras cosas, así como las versiones de SW que se han utilizado, versiones del lenguaje etc. En general se generan estas cosas de manera automática también.
* README.md: El readme que se utilizará para renderizar información en repositorios como GitHub.
* tsconfig.json: Le dice a TypeScript como trabajar, sobre qué versiones de ES trabajamos entre otras cosas.
* tslint.json: Nos fuerza a escribir código más limpio. Es un conjunto de reglas para la correcta codificación del código. Se suelen dejar las reglas establecidas por defecto.

## Utilizando Bootstrap 4 y creando un componente

### Bootstrap

Se trata de un pack de estilos CSnS y de componentes ya escritos que se pueden descargar libremente del siguiente enlace:

<https://getbootstrap.com/>

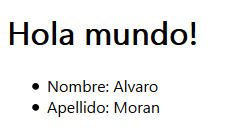
Existen muchos de estos agregadores de contenidos por internet para ser aplicados a un proyecto Angular. Básicamente proporcionan funcionalidades y componentes ya creados (botones, vistas de calendario, combo boxes…) con una serie de estilos predefinidos que se pueden aplicar, todo ello pensado para poder crear con rapidez una interfaz completamente funcional.

Se pueden descargar los archivos ya compilados o bien el código fuente. Otra forma de hacerlo, la que utilizaremos preferiblemente, será añadir Bootstrap como source para nuestro proyecto.

Para ello hay que ir a la sección BootstrapCDN de la página anterior y copiar el segmento XML que ahí se proporciona:

|  |
| --- |
| <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous"> |

Lo pegaremos en nuestro archivo index.html de nuestro código dentro de la cabecera y debajo de otro link que se ha generado por defecto. Esto automáticamente cambiará el estilo de nuestra aplicación:



Cuando queramos utilizar un elemento propio de Bootstrap podemos ir a la página web detallada anteriormente y realizar una búsqueda del componente que queramos utilizar, siendo después de ello tan solo necesario copiar y pegar el código que se adjunte en la página web sobre el componente deseado.

### Creación manual de componentes

Ahora creamos un componente y utilizaremos los estilos que acabamos de importar. Hay una forma automática de crear estos componentes, pero vamos a empezar haciéndolo de forma manual para ver qué archivos son los que se generan y son los que definen completamente el comportamiento de un nuevo componente. Crearemos una nueva carpeta en app llamada components, que donde por convenio se generan los componentes personalizados:

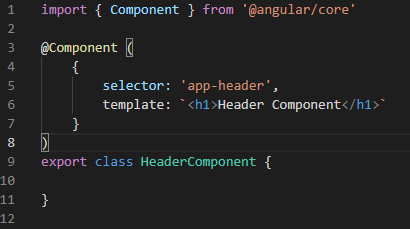


Este es el convenio que se suele utilizar:

* Carpeta components con los componentes personalizados.
* Carpeta header para componentes de la cabecera.
* Clase typescript donde indicamos que se trata de un componente mediante el nombre \*.component.ts. El component es opcional, pero se usa por convenio para identificar rápidamente componentes.

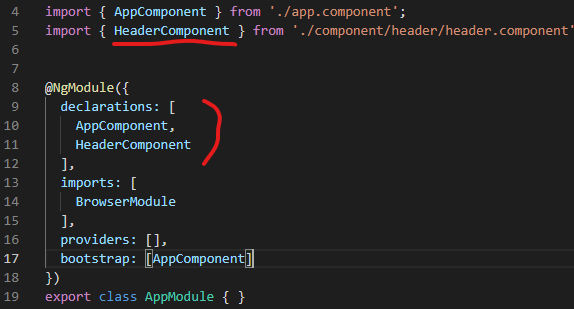
**NOTA: Por un bug entre distintos editores, parece que es necesario reiniciar determinados editores para que estas exportaciones sean compiladas correctamente.**

El contenido inicial de nuestro componente será el siguiente:

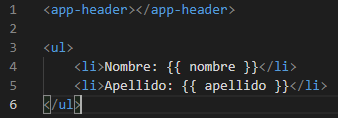


* Tenemos una clase definida HeaderComponent dentro, vacía de momento.
* Inidicamos que es un componente mediante el decorador Component, importado desde @angular/core.
* Definimos aspectos del componente:
  + **selector**: será el nombre utilizado para acceder o seleccionar este nuevo componente externamente, como a través del HTML como veremos más adelante.
  + **template**: La representación HTML del componente. Puede definirse en un archivo html independiente o bien como se ha detallado en la imagen anterior, directamente en el decorador. En la medida de lo posible se recomienda siempre utilizar un archivo HTML dedicado, si bien para el caso que nos ocupa donde el contenido es una línea de código HTML se puede especificar en el decorador sin problemas. En la siguiente sección se detallará como hacerlo en un archivo HTML dedicado.
  + **export**: Necesario para poder exportar la clase desde nuestra aplicación.

Para exportar el nuevo componente creado dentro de nuestra aplicación de Angular tenemos que modificar nuestro app.module.ts para incluirlo de la siguiente forma:



Finalmente, vamos a utilizar nuestro nuevo componente en el HTML de nuestra aplicación. Para ello modificamos el archivo app.component.html para referenciarlo de la siguiente forma:



## TemplateUrl: Separando el HTML del componente

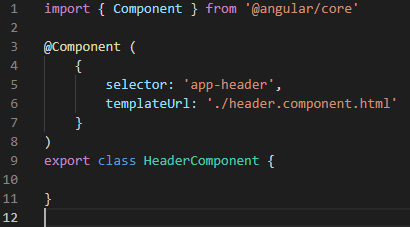
Para esta sección vamos a utilizar el componente Navbar que nos proporciona Boostrap. Para ello, en la web que se ha proporcionado en la sección anterior, podemos ir a la sección de documentación y realizar una búsqueda sobre ese componente. Nos aparecerá una sección donde está todo el código que genera la totalidad de dicho componente. Como se puede ver, se trata de un código bastante extenso y sería bastante ineficiente ponerlo dentro de la sección ‘template’ de nuestro componente.

Es más, es muy posible que nos de un warning el mismo editor de texto por que la longitud en caracteres dentro de ese campo excede un máximo establecido. Ese máximo se puede modificar en las reglas establecidas en tslint.json, en el apartado “max-line-length”.

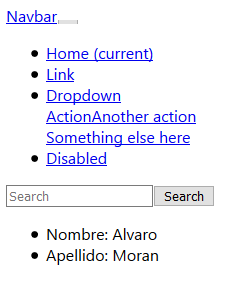
No obstante, haremos caso de la sugerencia de estilos e intentaremos pasar todo ese código a un archivo independiente para que quede más organizado.

Dentro de header crearemos un nuevo archivo llamado header.component.html (sigue siendo un estándar nombrarlo así). Dentro de ese nuevo archivo podemos pegar todo el código HTML que está asociado al componente (el código relativo a Navbar en este caso) para sacarlo de nuestra definición typescript del componente.

Ahora, para referenciar desde nuestro typescript el archivo donde está definido su template HTML tenemos que poner lo siguiente:



Como vemos, ahora el tag template ha cambiado por templateUrl, especificando la dirección (relativa) donde se encuentra el template en HTML relativo a este componente. Si ejecutamos la aplicación veremos lo siguiente (no tenemos relacionados importados así que se verá algo feo):



Podemos hacer lo mismo para definir un body o cualquier otro elemento que queramos de nuestra interfaz.

## Creando el Footer Component de manera automática

En esta sección veremos cómo se generan los componentes de forma automática, a diferencia de como hemos hecho hasta ahora.

Para ello, desde la terminal escribimos lo siguiente:

|  |
| --- |
| ng g c components/footer |

Desglosemos el comando:

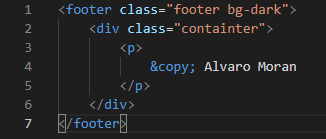
* Bandera g: Indica generate, para generar algo.
* Bandera c: Indica component, el objeto a generar será un componente.
* Path: Se especifica el path relativo en donde posicionaremos el componente. Tenemos el diretorio components creado, y al especificar components/footer se entenderá que queremos crear un nuevo componente footer con todos sus archivos asociados.

Se generan cuatro archivos automáticamente:

* Archivo css: Archivo que se aplicará solo al componente footer.
* Archivo spec.ts: Archivo de pruebas asociados a este componente.
* Archivo html: El template, la representación asociado a este archivo.
* Dwqdq

Cuando creamos el componente se genera automáticamente también dentro del app.module.ts, con lo que no tenemos que importarlo manualmente a nuestra aplicación.

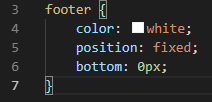
Ahora simplemente podemos pasar a definir el contenido de nuestro footer de la siguiente forma:



Y, posteriormente, lo tendremos que incluir en el HTML general de la aplicación de la siguiente forma:



Si quisiéramos definir los estilos de forma global, algo que es común, debemos utilizar el archivo styles.css que se genera automáticamente en cuanto creamos la aplicación en angular. De esta forma, por ejemplo, si en dicho archivo incluimos lo siguiente:

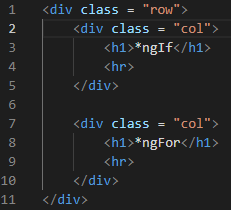


Y este estilo se aplicará de manera automática al componente footer que hemos creado sin necesidad de hacer nada más.

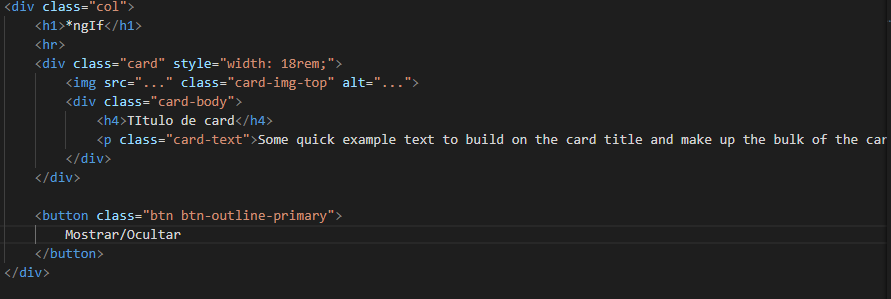
## Estructura del Body Component con directivas estructurales: \*ngIf y \*ngFor

En esta sección daremos al body de una cierta estructura. Para ello utilizaremos las Cards que nos proporciona bootstrap4. Se pueden utilizar como hemos hecho anteriormente, yendo a la web de Bootstrap, buscando el componente y copiando su código HTML dentro del HTML del componente body.

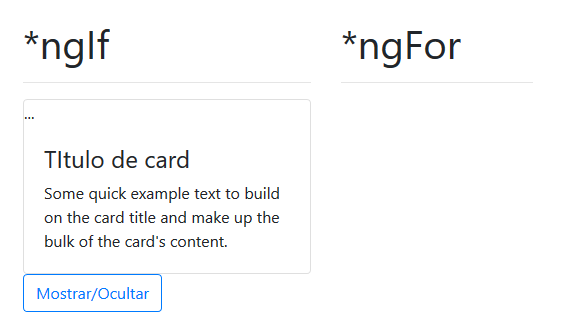
Antes de eso crearemos la siguiente estructura, con una fila y dos columnas que van a contener dos cards, cada uno con dos cabeceras que nos servirán para ilustrar los conceptos de ngIf y ngFor que vamos a ver en esta sección:



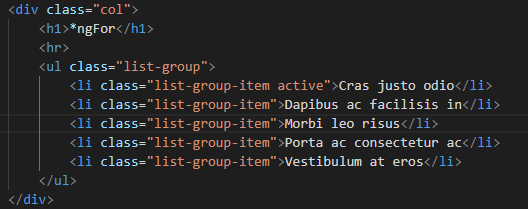
Después, debajo de la primera columna posicionaremos el código que hemos copiado para el card de Bootstrap y, además, un botón que utilizaremos más adelante:



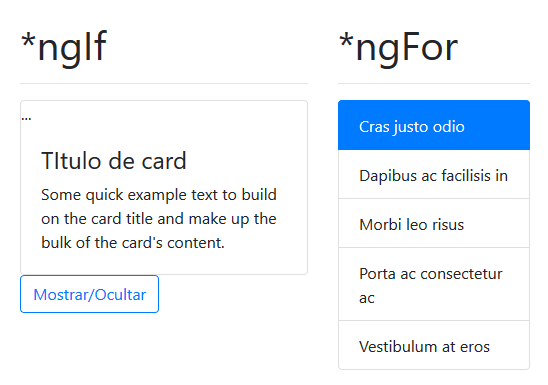
El resultado gráficamente será algo similar a esto, dependiendo del modelo de card que hayamos elegido:



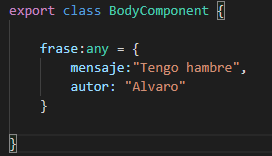
En la columna derecha introduciremos una lista. Bootstrap también nos proporciona listas de ejemplo, como la siguiente:



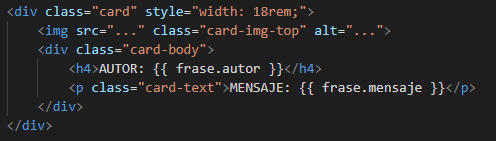
Gráficamente el resultado final que obtenemos es el siguiente:



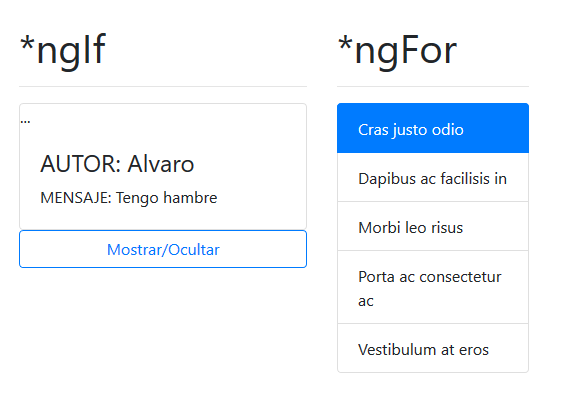
Vamos a crear un par de propiedades ahora dentro de nuestra clase BodyComponent, para que pueda escribir un mensaje y un autor:



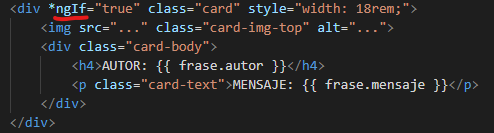
Y lo relacionamos desde el HTML para que nos muestre esto como contenido de la tarjeta en \*ngIf:



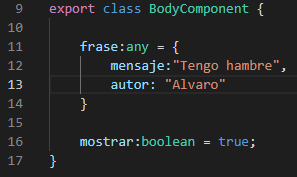
Gráficamente el resultado es el siguiente:



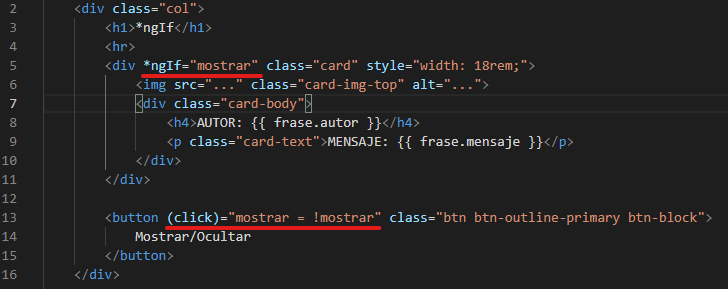
Ahora vamos a hacer que al pulsar el botón que hemos añadido se muestre o no el mensaje de la card. Para ello utilizaremos la directiva ngIf. Hay que resaltar que cuando no mostramos algo, Angular no oculta el elemento, sino que lo destruye del HTML. Debemos de colocar la directiva de la siguiente forma:



En este caso simplemente nos indicará que el elemento se mostrará siempre (el valor es true). Pero esto lo podemos relacionar con alguna variable de nuestra clase. Por ejemplo, supongamos que creamos el elemento mostrar:

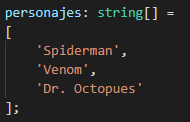


Una vez creada la variable, podemos modificar el HTML para que hagan referencia a ella y que en función del valor de ese booleano se muestre o no se muestre el componente:

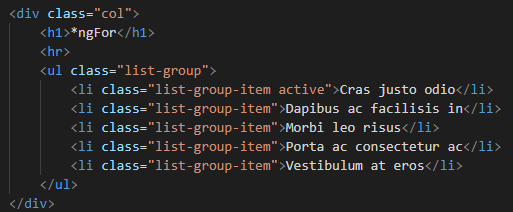


En este caso lo que hacemos es que cuando se llama al método click del botón (cuando se pulsa), cambiamos el valor del booleano por su opuesto. Más adelante veremos cómo se pueden hacer cosas más complejas llamando a métodos escritos en nuestro archivo Typescript. Por otro lado, la card se muestra o no en función del valor de la variable mostrar gracias a la directive \*ngIf.

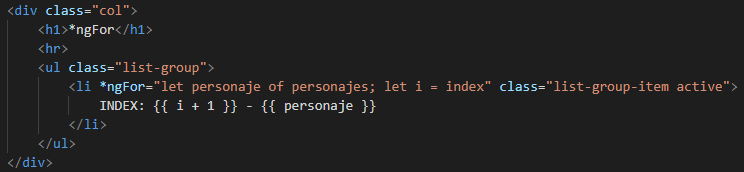
Ahora veremos un ejemplo de la directiva ngFor. Esta directiva sirve para mostrar elementos en manera de bucle; es ideal para representar arrays de elementos. Para ello, creamos la siguiente lista de elementos en nuestro typescript:



Ahora queremos mostrar esta lista gráficamente. Hasta ahora teníamos una lista en la que definíamos los elementos uno a uno, tal que así:

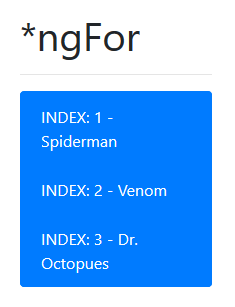


Sin embargo, lo que queremos es recorrer nuestra lista de personajes que acabamos de crear en nuestro Typescript. El elemento que se va a repetir es el list-group-item, esto es, deseamos crear un ítem para cada uno de los elementos que tenemos en nuestra lista; por tanto, aplicaremos la directiva \*ngFor a la clase list-group-item y dejaremos solo una. Tenemos que hacer un elemento que recorra nuestra lista de personajes. Este elemento lo declararemos junto al ngFor con un let, tal que así:



Además, hemos creado otra variable i que nos servirá para representar el índice por el cual vamos iterando. Nótese que el código entre “” es código javascript puro, pudiendo meter ahí dentro el código que queramos en ese lenguaje. Al igual que con la directiva ngIf, veremos formas de escribir esto de manera más elegante y mediante funciones dentro del Typescript. Pero de momento, servirá como ejemplo.

La representación gráfica de lo anterior es la siguiente:



# Aplicación de una página

## Iniciar el proyecto

Como se ha visto en la sección anterior, para crear un proyecto de Angular en una carpeta determinada (en este caso con nombre spa) introduciremos el siguiente comando:

|  |
| --- |
| ng new spa |

Para esta sección se proporcionan los siguientes recursos, que utilizaremos para representar estilos e imágenes:



## Creando la estructura del proyecto

Para crear nuevos componentes, utilizaremos los comandos que hemos visto anteriormente:

|  |
| --- |
| ng g c components/footer |

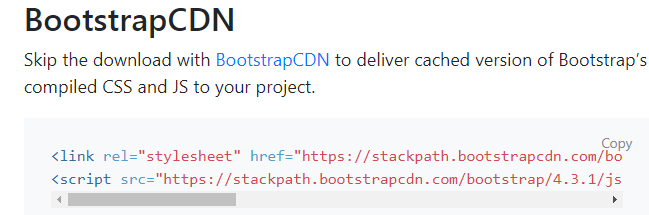
Si nos equivocamos al crear un componente y lo ponemos en una ruta incorrecta, con moverlo a mano y cambiar la referencia en el app.module.ts es suficiente para arreglar el problema. Ídem si queremos borrar el componente, habrá que modificar la información que le referencia en dicho archivo.

## Instalando Bootstrap y librerías de terceros

En esta sección veremos como instalar Bootstrap u otras librerías de terceros sobre nuestro proyecto de Angular.

### Mediante el CDN (Content Delivery Network)

Mediante este método lo único que tenemos que hacer es utilizar su CDN, su gestor de contenido. En Bootstrap, e igual para otras librerías de terceros, en su página web aparece el código que hay que importar a nuestro proyecto para que el gestor de contenido descargue automáticamente las dependencias:



Todos estos enlaces se ponen en el archivo index.html de nuestra aplicación.

Por convenio, los links se suelen incluir en la cabecera mientras que los scripts se incluyen en el body, debajo del app-root.

### Mediante descarga de las librerías

Otra forma de instalar las librerías externas es descargándolas directamente desde la web del proveedor. Estas librerías suelen venir en una carpeta de distribución (dist), dentro de las cuales están todos los JS necesarios para el Bootstrap. Este contenido se pondrá dentro de nuestra aplicación en la ruta src/assets/libs/<Nombre de librería como se desee>.

Una vez hecho esto, en el index.html tenemos que establecer un link que apunte a nuestras librerías que acabamos de importar, dentro de las cuales siempre hay un CSS genérico (es recomendable usar la versión min de los mismos).

<link rel="stylesheet" href="assets/libs/bootstrap/css/bootstrap.min.css">

Posteriormente, habría que descargarse también todos los scripts a utilizar y referenciarlos dentro del index.html.

### Utilizando el NPM

El Node Package Manager nos permite también descargar lo necesario para descargar e importar estas librerías de terceros.

Para hacer la instalación de las librerías de este modo, en las páginas de información de las mismas aparecerá el comando a utilizar con el npm para realizar la descarga.



Podemos añadir, como en el ejemplo anterior, el comando –save, que da a entender que se trata de una librería que nuestra aplicación va a necesitar, y los va a descargar y guardar en los módulos de NodeJS.

Al hacer esta instalación, veremos como dentro de la carpeta node\_modules del proyecto la librería de Bootstrap está ya incluida. Ahora tenemos que referenciar estos módulos desde nuestra aplicación; para ello, modificaremos el archivo angular.json, en el cual se describen parámetro de arranque de la aplicación, localización de los assets, etc. Este archivo tiene una sección de scripts y de styles en donde podremos hacer referencia a los node\_modules que nuestro NPM acaba de instalar. Un ejemplo gráfico de cómo realizar esta configuración es la siguiente:



Para aplicar estos cambios es necesario volver a ejecutar el ng serve.

## Configuración del NavBar y otros componentes

Para hacer la personalización de nuestros elementos necesitaremos los recursos proporcionados unos apartados atrás.

Las imágenes, por convenio, se pegan en la ruta siguiente:

src/assets/img

Con respecto al favicon, éste se encuentra directamente en la carpeta src de nuestra aplicación.

Para importar el NavBar, ya se vio en secciones anteriores como realizar su importación desde la web de Bootstrap. Si indagamos más dentro de la página de NavBar asociado a Bootstrap, veremos que además se nos proporciona la funcionalidad de asociar un icono en el navbar. Esto también nos interesará importarlo dentro del código. Así mismo, vamos creando otra serie de páginas que utilizaremos en nuestra aplicación, como el about y el propio home, siguiendo lo comentado en el apartado anterior.

## Rutas en Angular

La aplicación que estamos realizando se trata de una aplicación de una sola página. Para navegar entre sus distintas secciones introducimos el concepto de rutas.

Los archivos de rutas se crean, por convenio, dentro de la carpeta app directamente, y con el nombre routes.ts.

Su contenido es como sigue, suponiendo que tenemos una página llamada home:

import {RouterModule, Routes} from '@angular/router';

import { HomeComponent} from './components/home/home.component';

const APP\_ROUTES: Routes = [

    {path: 'home', component: HomeComponent},

    // Ruta por defecto

    {path: '\*\*', pathMatch:'full', redirectTo: 'home'}

];

export const app\_routing = RouterModule.forRoot(APP\_ROUTES);

Tenemos que importar este gestor de rutas dentro del árbol principal de nuestra aplicación de Angular. Para ello, en el archivo app.module.ts:

// Rutas

import { APP\_ROUTING } from './routes';

Como vemos, se suelen distinguir distintas secciones para importar distintas utilidades dentro del gestor de módulos de la aplicación:

* Rutas
* Servicios
* Componentes

Lo podemos organizar como queramos, pero será útil tener una buena organización en este elemento. Adicionalmente, tenemos que introducir las rutas dentro de los imports de nuestra aplicación (con lo anterior solo está disponible, hemos importado la librería pero no se usa de manera efectiva en ningún sitio):

@NgModule({

  declarations: [

    AppComponent,

    NavbarComponent,

    HomeComponent,

    AboutComponent,

    HeroesComponent

  ],

  imports: [

    BrowserModule,

    APP\_ROUTING

  ],

  providers: [],

  bootstrap: [AppComponent]

})

Adicionalmente, ya que hemos definido una ruta inicial por defecto home dentro de nuestro archivo de rutas, si lo incluimos dentro de nuestro app.component.html de la siguiente forma, veremos como carga esa ruta inicial home automáticamente:

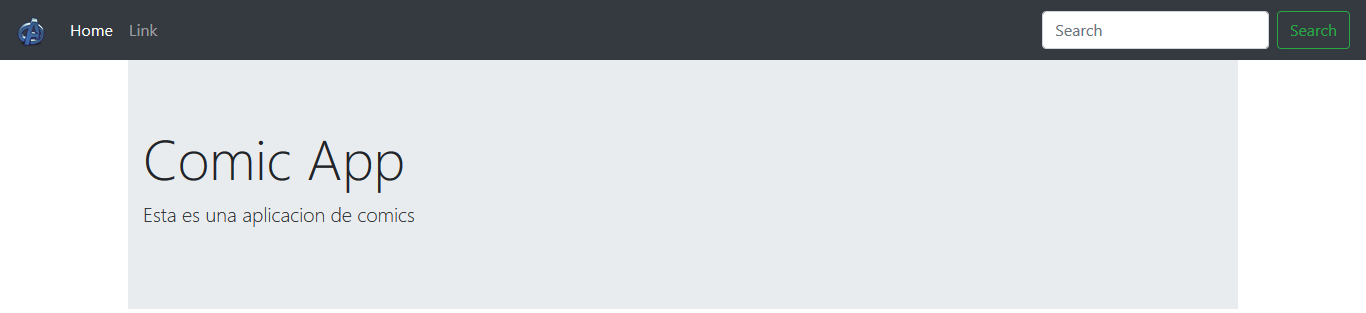
<app-navbar></app-navbar>

<div class="container">

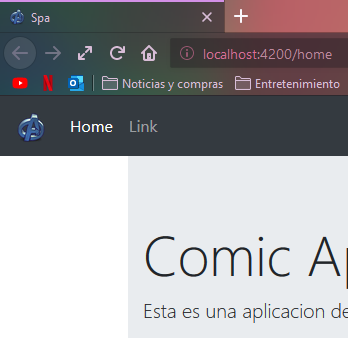
    <router-outlet></router-outlet>

</div>

El resultado es el siguiente:



En Angular hay distintas formas de asignar las rutas. Por ejemplo, tal como lo tenemos hecho ahora donde usamos path relativos sin caracteres raros:



Es una asignación directa, como si hubiera un folder home y estuviera accediendo a su index. Esto no es válido cuando pasamos parámetros, donde si hacemos un refrescar de la página podemos perder sus referencias y el contenido de la aplicación. Para evitarlo y tener nuestra aplicación preparada para ello, en nuestro archivo routes.ts tenemos que modificar la constante que hemos definido de la siguiente forma:

export const APP\_ROUTING = RouterModule.forRoot(APP\_ROUTES, {useHash:true});

Esta segunda forma es un poco más eficiente y está más soportado por los navegadores web, aunque habrá veces que se pierda el estado de igual forma. Se verá más adelante cómo solventar estos problemas. Para que funcione esta segunda forma con el hash es necesario tener la referencia de home incluida en el código, en el archivo index.html:

<base href="/">

## Routelink y Routelinkactive

Para probar la navegación entre páginas primeramente debemos añadir todas las páginas que deseemos dentro de nuestra constante APP\_ROUTES, que hemos definido en la sección anterior.

Una vez incluidas todas las páginas que necesite nuestra aplicación, nos iremos al componente navBar, que será el encargado de mostrar el menú de las posibles páginas de nuestra aplicación.

        <ul class="navbar-nav mr-auto">

            <li class="nav-item" routerLinkActive="active">

                <a class="nav-link" [routerLink]="['home']">Home</a>

            </li>

            <li class="nav-item" routerLinkActive="active">

                <a class="nav-link" [routerLink]="['heroes']">Heroes</a>

            </li>

            <li class="nav-item" routerLinkActive="active">

                <a class="nav-link" [routerLink]="['about']">About</a>

            </li>

        </ul>

Como vemos, hemos incluido la referencia a las páginas (su ruta) gracias al objeto routerLink, que tiene después un array con la relación de rutas necesarias para llegar al objeto. Por ejemplo, si queremos que una página en concreto esté en la ruta /home/1/2, debería tener lo siguiente:

<a class="nav-link" [routerLink]="['home', '1', '2']">Home</a>

El nombre del objeto (‘home’, en el caso anterior), es el nombre que hayamos especificado en el path dentro de nuestro routes.ts:

const APP\_ROUTES: Routes = [

    // Rutas existentes

    {path: 'home', component: HomeComponent},

    {path: 'about', component: AboutComponent},

    {path: 'heroes', component: HeroesComponent},

    // Ruta por defecto

    {path: '\*\*', pathMatch:'full', redirectTo: 'home'}

];

La sentencia routerLinkActive nos permite nos permite iluminar dentro de nuestro navBar el elemento de navegación que está activo en función del elemento de nuestro routedLink que esté activo. Angular, si detecta un routerLink, busca en el elemento padre automáticamente si tiene la sentencia routerLinkActive para poder realizar esta operación.

Supongamos que ahora queremos hacer una transición fluida entre las distintas páginas. Dentro de los recursos de la aplicación tenemos el archivo animate.css, que tiene una serie de estilos que nos ayudarán a ello. Copiaremos su contenido y lo pegaremos en el elemento de estilos general de la aplicación, styles.css.

## Componente heroes

Para representar los héroes en nuestra aplicación nos iremos a la web de Bootstrap y nos fijaremos en todas las tarjetas de contenido que nos puede ofrecer la librería. Importaremos una de ella y le agregaremos un botón en la parte inferior, para redirigir posteriormente a la página que despliega información adicional del héroe. Además, insertaremos una imagen (aunque de momento no será variable, la meteremos hard-codeada).

Todo ello en nuestro héroes.component.html:

<h1>Héroes <small>Marvel y DC</small></h1>

<div class="card-columns">

    <div class="card">

        <img class="card-img-top img-fluid" src="assets/img/aquaman.png" alt="Card">

        <div class="card-block">

            <h4 class="card-title">Card title</h4>

            <p class="card-text">This card has supporting text below as a natural lead-in to additional content.</p>

            <p class="card-text"><small class="text-muted">Last updated 3 mins ago</small></p>

            <button type="button" class="btn btn-outline-primary btn-block">

                Ver más...

            </button>

        </div>

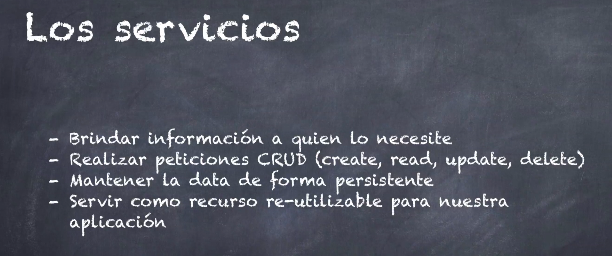
    </div>

</div>

## Servicios

### Creando nuestro primer servicio

Los servicios nos permiten que, con un mismo componente de héroes, podemos modificar su contenido sin modificar su lógica de negocio, de forma que nos ahorramos escribir código. Los servicios nos permiten:



Su funcionamiento lo podemos resumir de la siguiente forma:

1. Un componente solicita a un servicio datos determinados.
2. El servicio devuelve información y el componente sigue trabajando.
3. Otro componente viene a preguntarle por más información.
4. El servicio devuelve información y el componente sigue trabajando.
5. Llega un componente que necesita actualizar los datos.
6. El servicio se encarga de preguntar al servidor y enviarle los datos actualizados al componente.

El servicio al fin y al cabo, nos permite compartir información a lo largo de toda la aplicación, sin ocuparse de cuestiones de maquetado ni nada por el estilo.

### Creando nuestro primer servicio y ngOnInit

El convenio dice que los servicios se localizan dentro de nuestra carpeta app, en una nueva carpeta denominada servicios o services, y con extensión \*.service.



El contenido del servicio inicialmente será lo siguiente:

import { Injectable } from '@angular/core';

@Injectable()

export class HeroesService{

    constructor() {

        console.log("Servicio listo para usar");

    }

}

Posteriormente, en nuestro app.module.ts, añadiremos nuestro servicio para que esté disponible para el resto de la aplicación:

// Servicios

import { HeroesService } from './services/heroes.service';

Dentro de todas las categorías de nuestra aplicación, le corresponde ser añadido en la sección providers:

  providers: [

    HeroesService

  ],

Adicionalmente, en el archive responsable de la página que gestionará el servicio (héroes.component.ts), tendremos que importar el servicio y utilizar el servicio en su constructor:

import { Component, OnInit } from '@angular/core';

import { HeroesService } from '../../services/heroes.service'

@Component({

  selector: 'app-heroes',

  templateUrl: './heroes.component.html'

})

export class HeroesComponent implements OnInit {

  constructor(private \_heroesService:HeroesService) {

  }

Por supuesto, otra forma equivalente de implementar el constructor de forma más implícita será la siguiente:

export class HeroesComponent implements OnInit {

  private \_heroesService;

  constructor(heroesService:HeroesService) {

    this.\_heroesService = heroesService;

  }

De esta forma el servicio se inicializará y quedará disponible en nuestro componente. Dentro de lo que es el servicio en sí, y ya que no disponemos de un servicio de backend que nos proporcione esa información, crearemos una variable donde guardaremos toda la información de los héroes y, además, un método público que nos permitirá acceder a los mismos.

Dentro del componente heroes.component.ts, que implementa onInit, vemos que también hay un método creado automáticamente que se llama ngOnInit; este método se llama una vez que se ha inicializado la clase por completo y está lista para ser utilizada. Es dentro de esta clase donde realizaremos el procesamiento necesario para cargar toda la información de los héroes, una vez que la página y sus subcomponentes han sido correctamente inicializados.

  public heroes:any[] = [];

  constructor(heroesService:HeroesService) {

    this.heroesService = heroesService;

  }

  ngOnInit() {

    this.heroes = this.heroesService.getHeroes();

  }

Lo ideal será que en vez de ser un array de tipo any sean un array de alguna interfaz que tenga sentido dada la información guardada, pero en este caso no nos centraremos en esas cuestiones de sintaxis de typescript.

Con este servicio así declarado ya estamos proporcionando la información de datos a nuestra página de héroes para que sean representados. Básicamente los servicios sirven de puente entre la interfaz meramente dicha y la forma de obtener o procesar los datos que en ella se representan.

## Diseño con \*ngFor

Ya tenemos todos los datos cargados en nuestro componente de héroes y tenemos, además, la interfaz con una tarjeta como para poder representar dichos elementos. Tendremos que modificar la interfaz que vimos hace dos apartados para representar, en forma de bucle, todos los datos que hemos cargado con nuestro servicio anterior. Para ello utilizamos la directiva \*ngFor como vimos en algunos capítulos atrás:

<h1>Héroes <small>Marvel y DC</small></h1>

<div class="card-columns">

    <div class="card" \*ngFor="let heroe of heroes">

        <img class="card-img-top img-fluid" [src]="heroe.img" [alt]="heroe.nombre">

        <div class="card-block">

            <h4 class="card-title">{{heroe.nombre}}</h4>

            <p class="card-text">{{heroe.bio}}</p>

            <p class="card-text"><small class="text-muted">{{heroe.aparicion}}</small></p>

            <button type="button" class="btn btn-outline-primary btn-block">

                Ver más...

            </button>

        </div>

    </div>

</div>

Merece la pena comentar una serie de cosas:

* El ngFor funciona como hemos visto anteriormente; llamamos a cada instancia particular de heroes como heroe, y esto recorre la lista llamada heroes del componente (por ello lo hemos hecho público, sino no sería acceisble por el HTML).
* Para cada imagen de cada heroe, hacemos referencia al valor interno de su objeto img. Para ello accedemos como si fuera Java, esto es, heroe.img. El parámetro src lo ponemos entre corchetes, lo cual indica en el código que esa porción de código estará gestionada por Angular.
* De igual forma, todas las llamadas a la hora de definir el comportamiento de un componente, las hacemos entre […] para indicar que lo gestione Angular (está bastante bien que sea así ya que el bucle se está recorriendo gracias a la directiva ngFor de Angular).
* Dentro de lo que sería el texto de los párrafos, esto es, su valor, accedemos a los componentes deseados mediante {{…}}.

## Rutas con parámetros

### Mediante Anchors

Vamos a crear un componente nuevo llamado heroe, que creará una nueva página para mostrar la información detallada de cada héroe.

Además de crearla la introduciremos en nuestro componente de rutas, donde ya estableceremos que su path será relativo dependiendo del id del héroe en cuestión:

import { HeroeComponent} from './components/heroe/heroe.component';

const APP\_ROUTES: Routes = [

    // Rutas existentes

    {path: 'home', component: HomeComponent},

    {path: 'about', component: AboutComponent},

    {path: 'heroes', component: HeroesComponent},

    {path: 'heroe/:id', component: HeroeComponent},

    // Ruta por defecto

    {path: '\*\*', pathMatch:'full', redirectTo: 'home'}

];

Posteriormente crearemos el botón, en esta ocasión no como un botón, sino como un anchor, un mero enlace a otras páginas al que le daremos el estilo de botón:

<a [routerLink]="['/heroe',i]" class="btn btn-outline-primary">Ver más...</a>

A la hora de crear el botón tenemos que tener en cuenta que routerLink genera el path concatenando su posición actual y lo que le añadimos. En el caso anterior, en el cuál nos encontramos dentro de la página heroes, si pulsamos un botón y tenemos definido el routerLink de la siguiente forma:

<a [routerLink]="['heroe',i]" class="btn btn-outline-primary">Ver más...</a>

Lo que sucederá es que la URL a la que accederemos será heroes/heroe/1. No es lo que queremos, dado que hemos definido que el path para nuestras páginas de héroes singular es heroes/1, directamente. Por ello es por lo que hay que ponerle barra / antes de heroe, para indicarle que partimos de la raíz de la aplicación para crear ese enlace.

Hemos visto que le estamos pasando el índice i que marca qué héroe de nuestro array de héroes estamos consultando. Para ello tenemos que añadir una sentencia a nuestra directiva ngFor que nos permitirá obtener el índice, de la siguiente forma:

<div class="card animated fadeIn fast" \*ngFor="let heroe of heroes; let i = index">

### Mediante funciones TypesScript

Si queremos utilizar un componente botón en lugar del anchor, haremos lo siguiente:

            <button (click)="verHeroe(i)" type="button" class="btn btn-outline-primary btn-block">

            Ver más...</button>

Como vemos, llamamos a la función verHeroes, que tendremos que declarar dentro del código TypeScript de nuestro componente. Básicamente, lo que haremos dentro de él será hacer la llamada al routerLink, en esta ocasión de otra forma por código; necesitaremos el objeto router como argumento de entrada (se lo podemos pasar en el constructor a nuestro componente) y luego utilizarlo para generar las rutas y acceder a ellas:

export class HeroesComponent implements OnInit {

  private heroesService:HeroesService;

  private router:Router;

  public heroes:any[] = [];

  constructor(heroesService:HeroesService, router:Router) {

    this.heroesService = heroesService;

    this.router = router;

  }

  // Carga la informacion del servicio una vez esta inicializado el componente

  ngOnInit() {

    this.heroes = this.heroesService.getHeroes();

  }

  verHeroe(id:number) {

    this.router.navigate( ['/heroe', id] );

  }

}

## Recibiendo parámetros por URL

### Recibiendo el parámetro

En el apartado anterior aprendimos a pasar un índice mediante una ruta, que nos permite acceder a cada ruta concreta de héroe. Utilizaremos este índice, que forma parte de la URL como argumento de entrada para nuestra página de héroe singular, de la siguiente forma:

import { ActivatedRoute } from '@angular/router';

export class HeroeComponent implements OnInit {

  private activatedRoute:ActivatedRoute;

  constructor(activatedRoute:ActivatedRoute) {

    this.activatedRoute = activatedRoute;

    this.activatedRoute.params.subscribe( params => {

      console.log(params);

    })

  };

  ngOnInit() {

  }

}

En este código nos estamos suscribiendo al observador que observa los parámetros de entrada, por así decir, que tiene la URL. Más adelante veremos más en detalle cómo funcionan los observadores y qué implica suscribirnos a ellos.

Lo anterior sirve para recibir cualquier tipo de parámetros. En este caso, que sabemos que simplemente vamos a recibir un ID podemos poner lo siguiente:

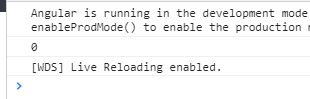
    this.activatedRoute = activatedRoute;

    this.activatedRoute.params.subscribe( params => {

      console.log(params ['id']);

    });

De esta forma por consola se nos escribe directamente el ID que recibimos parseado correctamente:



Lo que escribimos dentro de las llaves [ ] tiene que ser igual a lo que especificamos en nuestro archivo de rutas:

{path: 'heroe/:id', component: HeroeComponent},

### Obteniendo información con el parámetro

Ahora que ya sabemos cómo recibir un parámetro, modificamos nuestra clase HeroeComponent para incluir también el servicio de obtención de datos de héroes y utilizarlo para obtener la información a ser representada en la página:

export class HeroeComponent implements OnInit {

  private activatedRoute:ActivatedRoute;

  private heroesService:HeroesService;

  public heroe:any = {};

  constructor(activatedRoute:ActivatedRoute, heroesService:HeroesService) {

    this.activatedRoute = activatedRoute;

    this.heroesService = heroesService;

    this.activatedRoute.params.subscribe( params => {

      this.heroe = this.heroesService.getHeroe(params ['id']);

    });

  }

Adicionalmente, tenemos que crear dentro del servicio el método que nos devuelva un héroe en base a un id concreto:

    getHeroe(index:string) {

        return this.heroes[index];

    }

## Introducción a los Pipes

Los pipes sirven para transformar los datos en la parte visual solamente, solo en el template que mostramos al usuario. Esto no afecta a los datos. Supongamos que queremos mostrar, por ejemplo, el nombre de nuestros héroes de distinta forma a cómo están almacenados (todos en mayúsculas, por ejemplo).

Información acerca de ellos podemos encontrarla en la web de documentación de angular. Más adelante veremos una sección sobre ellos más en detalle.

<https://angular.io/guide/pipes>

Básicamente, utilizando una pipe (‘|’) podemos modificar el dato que vamos a mostrar. Por ejemplo, si queremos que el nombre del héroe aparezca en mayúsculas usaremos:

<h1> {{ heroe.nombre | uppercase}} <small>Aparición: {{ heroe.aparicion | date:'y'}}</small></h1>

En el ejemplo anterior vemos además como, para el pipe date, podemos pasar un argumento a un pipe mediante el carácter ‘:’.

## Funcionalidad de búsqueda

La funcionalidad de búsqueda nos va a permitir ilustrar cómo pasar argumentos desde la parte gráfica a la lógica, en este caso, mediante un cuadro de búsqueda.

Para ello necesitamos dos cosas:

* Asignar un alias al elemento del cuál obtendremos la información
* Asignar una acción cuando clickeemos o realicemos la opción que deseemos que lance el proceso de búsqueda.

Lo vemos en el siguiente ejemplo:

<input class="form-control mr-sm-2" type="search" placeholder="Buscar héroe" (keyup.enter)="buscarHeroe(buscarTexto.value)" #buscarTexto>

Aquí hemos asociado al dejar de pulsar la tecla intro una función (que describiremos a continuación), y como argumento le pasamos el valor de buscarTexto, donde buscarTexto es un alias (mediante el carácter #) que le hemos puesto a la caja de búsqueda.

Tenemos que hacer un método buscarHeroes también en el servicio que nos da la información, de tal forma:

    buscarHeroes(termino:string):Heroe[] {

      let heroesArr:Heroe[] = [];

      termino = termino.toLowerCase();

      // Recorremos la lista de heroes

      for (let heroe of this.heroes) {

        let nombre = heroe.nombre.toLowerCase();

        if (nombre.indexOf(termino) >= 0) { // busca string dentro del nombre

          heroesArr.push(heroe);

        }

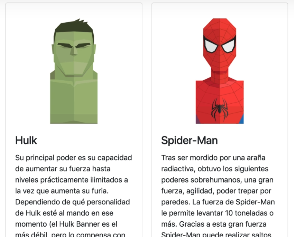
      }

      return heroesArr;

    }

## Recibir información de componente padre a hijo con @Input

Imaginemos que queremos crear un componente especializado que sea una de las tarjetas que estamos observando actualmente, de manera que podamos reutilizarlas cuando listamos todos los héroes o cuando realizamos búsquedas:



Uno de los problemas que teníamos hasta el apartado anterior era que teníamos que duplicar código para mostrar los resultados en la ventana de búsqueda y en la ventana de listar héroes (duplicábamos el código de cómo se representan las tarjetas).

Crearemos un nuevo componente para ello, llamado heroeTarjeta, que debe utilizar el componente Input de Angular, de la siguiente forma:

import { Component, OnInit, Input } from '@angular/core';

import { Heroe } from '../../services/heroes.service';

@Component({

  selector: 'app-heroe-tarjeta',

  templateUrl: './heroe-tarjeta.component.html'

})

export class HeroeTarjetaComponent implements OnInit {

  @Input() heroe:Heroe;

  constructor() { }

  ngOnInit() {

  }

}

Con este decorador habilitamos que cuando utilicemos la etiqueta HTML que llama a nuestro nuevo componente le podamos pasar argumentos, de la siguiente forma:

<app-heroe-tarjeta [heroe]="heroe" \*ngFor="let heroe of heroes; let i = index"></app-heroe-tarjeta>

El nombre de parámetro [heroe] podría ser cualquiera, siempre que sea igual al nombre de la propiedad declarada dentro del código TypeScript. Y le asignamos el valor del heroe actual que está siendo recorrido en el ngFor.

Para saber qué tarjeta tenemos que representar, nuestro nuevo componente necesitará también un índice, el índice que solíamos utilizar para obtener los datos. Nos queda entonces de la siguiente forma su llamada:

<app-heroe-tarjeta [heroe]="heroe" [index]="i" \*ngFor="let heroe of heroes; let i = index"></app-heroe-tarjeta>

Y el contenido final de nuestro componente tarjeta será como sigue:

export class HeroeTarjetaComponent implements OnInit {

  @Input() heroe:Heroe;

  @Input() index:Number;

  constructor(private router:Router) { }

  ngOnInit() {

  }

  verHeroe() {

    this.router.navigate( ['/heroe', this.index] );

  }

}

## Comunicación de hijo a padre con @output

Supongamos que, a diferencia de lo que hemos visto en el apartado anterior, no queremos implementar dentro de la funcionalidad de las tarjetas el método verHeroe para navegar, sino que queremos que eso lo gestione el padre, esto es, la página que hace uso de las tarjetas que hemos creado. Tenemos que establecer una comunicación entre cada tarjeta particular y el padre. Para ello, tenemos que utilizar el decorador output, con la ayuda de event emitter:

import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';

import { Heroe } from '../../services/heroes.service';

import { Router } from '@angular/router';

@Component({

  selector: 'app-heroe-tarjeta',

  templateUrl: './heroe-tarjeta.component.html'

})

export class HeroeTarjetaComponent implements OnInit {

  @Input() heroe:Heroe;

  @Input() index:number;

  @Output() heroeSeleccionado:EventEmitter<number>;

  constructor(private router:Router) {

    this.heroeSeleccionado = new EventEmitter();

  }

  ngOnInit() {

  }

  verHeroe() {

    this.heroeSeleccionado.emit(this.index);

  }

}

Como vemos en el caso anterior, hemos declarado un Output, una salida, a la que hemos llamado heroeSeleccionado y le hemos dado el tipo EventEmitter, esto es, es un elemento observable, un evento que puede ser observado por otros. Le decimos, además, que lo que va a emitir una vez que el evento se lance será una variable de tipo number. Finalmente, en el método verHeroe, emitimos la información (recordemos que este método se llamaba cuando pulsábamos el botón). Esto se lo que sucede, por ejemplo, cuando tenemos un botón; los botones tienen un Output que es click.

Ahora tenemos que hacer que el padre reciba este evento que es emitido por la tarjeta que está representando bajo él. Para ello, y como hacemos con un botón o cualquier otro evento de una clase que hayamos utilizado hasta ahora:

<app-heroe-tarjeta [heroe]="heroe" [index]="i" (heroeSeleccionado)="verHeroe( $event )" \*ngFor="let heroe of heroes; let i = index"></app-heroe-tarjeta>

Donde, como vemos, ahora le estamos pasando cono inputs el índice y héroe seleccionado y recibimos como output (o esperamos a recibirlo, más bien) el evento de tipo heroeSeleccionado. Esto, a su vez, llama a la función verHeroe del padre, que es el encargado de hacer la navegación. El argumento que necesita el padre (el índice, si recordamos, como vemos en el código más abajo) es el índice de la tarjeta a representar; hemos especificado que heroeSeleccionado devuelve un number que va a ser ese índice, y a ello accedemos mediante el comando **$event** dentro del HTML.

  verHeroe(index:number) {

    this.router.navigate( ['/heroe', index] );

  }

# Pipes

### Introducción

Los pipes nos permiten transformar los datos solo visualmente de un tipo de representación a otro (sin modificar el contenido que se haya proporcionado por el backend). Esto puede ser útil para mostrar la información de forma más visual e incluso para tareas más particulares, como la de mostrar una contraseña con asteriscos.

En este apartado veremos una serie de los pipes más comunes y veremos cómo crear pipes personalizados. En resumen:

* Pipes uppercase y lowercase
* Pipe Slice
* Pice Decimal
* Pipe Percent
* Pipe Currency
* Pipe Json
* Pipe Async
* Pipe Date
* Pipes personalizados
  + Capitalizar palabras y nombres
  + Creación de un pipe, que permite cargar recursos externos de forma segura.

Los pipes tienen ese nombre debido a que se utiliza el carácter pipe, muy propio de shells de Unix, por ejemplo, esto es: ‘|’.

### Repaso de los tipos de Pipes más comunes

Para este apartado crearemos un proyecto particular donde veremos cómo se comportan los distintos pipes que iremos viendo.

### UpperCase y LowerCase

Pipes para poner cadenas en mayúsculas o minúsuclas. El código de ejemplo es como sigue:

            <tr>

                <td>{{ nombre }}</td>

                <td>uppercase</td>

                <td>{{ nombre | uppercase}}</td>

            </tr>

            <tr>

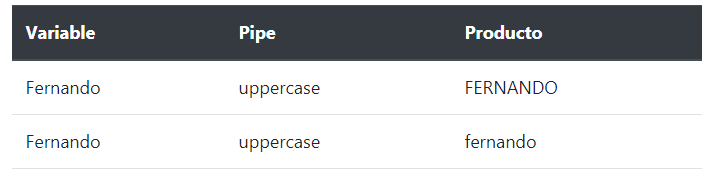
                <td>{{ nombre }}</td>

                <td>uppercase</td>

                <td>{{ nombre | lowercase}}</td>

            </tr>

Que, gráficamente, devuelve:



### Slice

Nos permite cortar un array, una cadena de texto o similares. Por ejemplo:

            <tr>

                <td>{{ nombre }}</td>

                <td>slice:3</td>

                <td>{{ nombre | slice:3 }}</td>

            </tr>

            <tr>

                <td>{{ nombre }}</td>

                <td>slice:0:3</td>

                <td>{{ nombre | slice:0:3 }}</td>

            </tr>

            <tr>

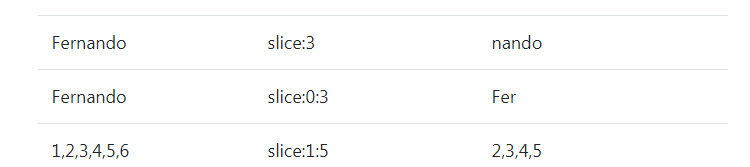
                <td>{{ array }}</td>

                <td>slice:1:5</td>

                <td>{{ array | slice:1:5 }}</td>

            </tr>

Cuyo resultado es el siguiente:



Además, podemos aplicar slices dentro de código y no solo para hacer representaciones. Por ejemplo, tenemos un bucle con \*ngFor y queremos recorrer una parte determinada de ese bucle, no todos los posibles valores del mismo:

        <ul>

            <li \*ngFor="let item of array | slice:3:20"> {{ item }}</li>

        </ul>

### Decimal

Este pipe nos permite mostrar los números de forma que tengan un formato en particular. El detalle de cómo funciona este pipe viene dado en la web siguiente:

<https://angular.io/api/common/DecimalPipe>

Podemos acotar la cantidad de dígitos decimales que queremos mostrar. Supongamos el siguiente ejemplo:

            <tr>

                <td>{{ pi }}</td>

                <td>number:'2.1-5'</td>

                <td>{{ pi | number:'2.1-5' }}</td>

            </tr>

Aquí estamos indicando que queremos que el número a representar tenga 2 números enteros y un número de entre 1 y 5 decimales, si los tiene. Si el número no tiene decimales se representará con cero hasta llegar al número mínimo de decimales.

### Percent: Porcentajes

El soporte para porcentajes no está desarrollado para todos los navegadores, es necesario que tengan soporte. No obstante, prácticamente cualquier navegador moderno tiene soporte para la representación de porcentajes.

Sea el siguiente código, por ejemplo:

            <tr>

                <td>{{ decimal }}</td>

                <td>percent</td>

                <td>{{ decimal | percent }}</td>

            </tr>

            <tr>

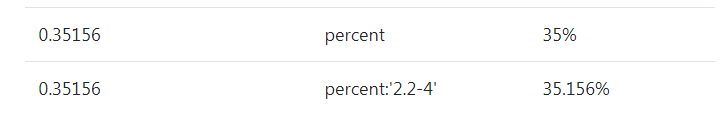
                <td>{{ decimal }}</td>

                <td>percent:'2.2-4'</td>

                <td>{{ decimal | percent:'2.2-4' }}</td>

            </tr>

El resultado es el que sigue:



### Currency: Monedas

El Pipe de currency es un poco más trabajado que el resto de pipes que hemos visto a continuación. Es recomendable ver la documentación de Angular al respecto:

<https://angular.io/api/common/CurrencyPipe>

Nos permite mucha personalización, a nivel del número de enteros y decimales que queremos, si queremos mostrar el código de la moneda (según el ISO 4217) y demás.

A nivel de código funciona de la siguiente manera:

            <tr>

                <td>{{ salario }}</td>

                <td>currency</td>

                <td>{{ salario | currency }}</td>

            </tr>

            <tr>

                <td>{{ salario }}</td>

                <td>currency: 'EUR':false</td>

                <td>{{ salario | currency: 'EUR':false }}</td>

            </tr>

            <tr>

                <td>{{ salario }}</td>

                <td>currency: 'EUR':true</td>

                <td>{{ salario | currency: 'EUR':true }}</td>

            </tr>

            <tr>

                <td>{{ salario }}</td>

                <td>currency: 'EUR':true:'3.4-6'</td>

                <td>{{ salario | currency: 'EUR':true:'3.4-6' }}</td>

            </tr>

            <tr>

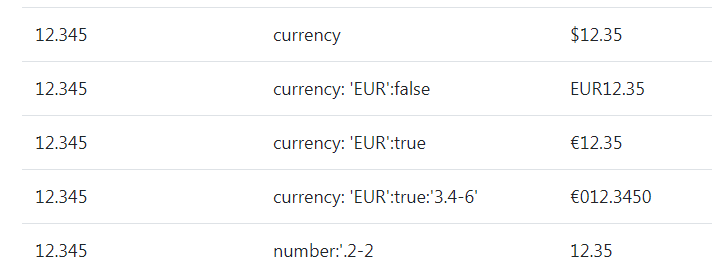
                <td>{{ salario }}</td>

                <td>number:'.2-2</td>

                <td>{{ salario | number:'.2-2' }}</td>

            </tr>

Que tiene la siguiente representación gráfica:



Vemos como el pipe muestra por defecto la cifra en dólares, salvo que le especifiquemos lo contrario. Además, si no queremos mostrar la moneda es posible que, como en el último caso, nos interese más aplicar un pipe de tipo number. Podemos especificar solo el número de decimales si queremos, sin modificar la parte entera con el pipe, como se ve en el ejemplo anterior.

### Pipe JSON

Es muy útil para representar información de objetos. Supongamos que tenemos un objeto como el que sigue:

  heroe = {

    nombre: "Logan",

    clave: "Wolverine",

    edad: 500,

    direccion: {

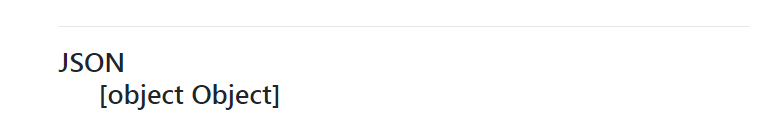
      calle: "Primera",

      casa: "3"

    }

  }

Si simplemente intentamos imprimir ese objeto por pantalla desde el HTML veremos lo siguiente:



Si le aplicamos el pipe Json, sin mayor argumento a la hora de imprimirlo por pantalla, tal que así:

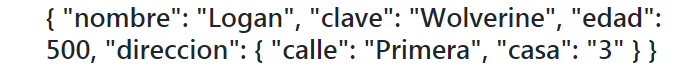
Obtenemos la siguiente imagen:

<ul>{{ heroe }}</ul>

Si le aplicamos de manera sencilla el pipe Json obtenemos lo siguiente:

<ul>{{ heroe | json }}</ul>

Con la representación siguiente:



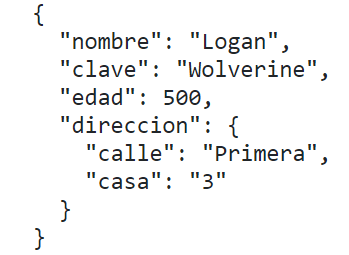
Adicionalmente, podemos introducir saltos de líneas con la ayuda del bootstrap4 y la etiqueta prev, como se ve en el siguiente código:

        <pre>

            <ul>{{ heroe | json }}</ul>

        </pre>

Cuya representación es la siguiente:



### Pipe Async

El pipe asíncrono nos permite mostrar información a partir de observables o promesas. Supongamos que tenemos el siguiente valor de promesa:

  valorDePromesa = new Promise ( (resolve, reject) => {

    setTimeout( ()=> resolve('Llego el dato'), 3500);

  });

Y aplicamos en el HTML el código siguiente:

            <tr>

                <td>{{ valorDePromesa }}</td>

                <td>async</td>

                <td>{{ valorDePromesa | async }}</td>

            </tr>

Lo que hace el pipe async en este caso es esperar al valor de resolve o reject de la promesa para intentar imprimirlo por pantalla. Una vez que llega el resultado de la misma, lo imprime en el momento adecuado:



## Importancia del locale

El DatePipe que utilizaremos en este ejemplo tiene muchas formas de representación diferentes, por lo que es recomendado mirar la documentación de Angular al respecto:

<https://angular.io/api/common/DatePipe>

Veamos un par de ejemplos de código de cómo se utiliza el DatePipe según está por defecto:

            <tr>

                <td>{{ fecha }}</td>

                <td>date</td>

                <td>{{ fecha | date }}</td>

            </tr>

            <tr>

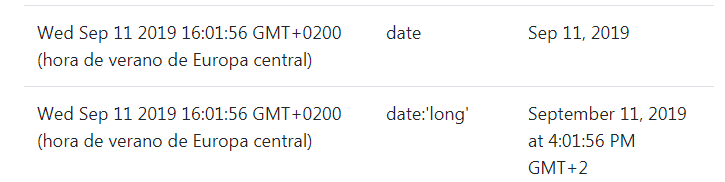
                <td>{{ fecha }}</td>

                <td>date:'long'</td>

                <td>{{ fecha | date:'long' }}</td>

            </tr>

Cuya representación gráfica es la siguiente:



Pero claro, nos está representando el número de días y horas en inglés, y nosotros deseamos que esta representación sea en castellano. Para ello tenemos que modificar el comportamiento del pipe cambiando el locale, esto es, la variable que modifica en qué localización estamos ejecutando la aplicación. Para ello, en el app.module.ts:

import { BrowserModule } from '@angular/platform-browser';

import { NgModule, **LOCALE\_ID** } from '@angular/core';

import { AppComponent } from './app.component';

import **localeEs** from '@angular/common/locales/es';

import { **registerLocaleData** } from '@angular/common';

**registerLocaleData(localeEs);**

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [ {

**provide: LOCALE\_ID, useValue: "es"**

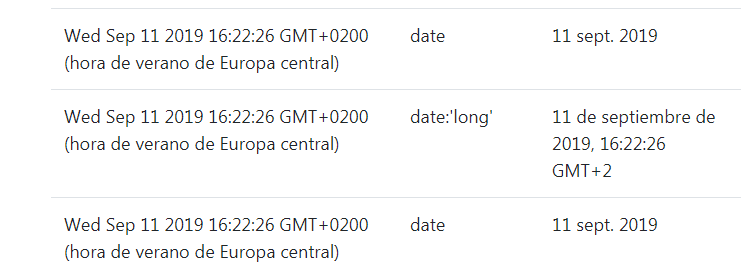
  }],

  bootstrap: [AppComponent]

})

export class AppModule { }

Con ello, el resultado es el siguiente:



Si no aplicamos el pipe la representación sigue siendo la misma. Sin embargo, si aplicamos el pipe date como hemos visto en el código más arriba, la representación se modifica siguiendo las directrices de la localización española a través del pipe.

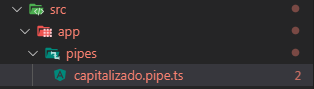
## Creación de pipes

### Primer ejemplo: Capitalizador

Supongamos que queremos crear un Pipe que represente nombres de personas con la primera letra en mayúscula y lo demás en minúscula y, supongamos que partimos del siguiente array de nombres:



Crearemos un pipe para formatear esta representación. Angular tiene su propio mecanismo para crear pipes, pero de momento lo haremos a mano. Creamos, para ello, una carpeta pipes dentro de app.



Debemos importar

La definición de código de pipe es como sigue:

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

    name: 'capitalizado'

})

export class CapitalizadoPipe implements PipeTransform {

    transform( value:string ) : string {

        value = value.toString().toLowerCase();

        // Separamos parabras

        let nombres = value.split(",");

        for (let i in nombres) {

            nombres[i] = nombres[i][0].toUpperCase() + nombres[i].substr(1);

        }

        return nombres.join(",");

    };

}

Debemos importarlo también dentro del app.module.ts de la siguiente forma, como si se tratase de un componente normal:

import  { **CapitalizadoPipe**} from './pipes/capitalizado.pipe';

registerLocaleData(localeEs);

@NgModule({

  declarations: [

    AppComponent,

**CapitalizadoPipe**

  ],

  imports: [

    BrowserModule

  ],

  providers: [ {

    provide: LOCALE\_ID, useValue: "es"

  }],

  bootstrap: [AppComponent]

})

Entonces, ahora podemos aplicar el pipe en el HTML como si se tratara de un pipe normal proporcionado por Angular, de la siguiente forma:

            <tr>

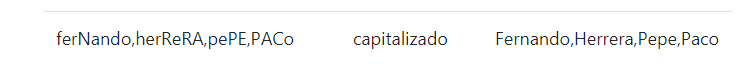
                <td>{{ nombres }}</td>

                <td>capitalizado</td>

                <td>{{ nombres | capitalizado }}</td>

            </tr>

Cuyo resultado gráfico es el siguiente:



El pipe puede recibir argumentos de entrada, como hemos visto en otros ejemplos. Veremos cómo podemos hacer esto en siguientes ejemplos con pipes.

### DomSeguro

# ANEXO

## A1. Manejo básico de estilos

Para asociar elementos con su estilo en concreto que le queramos aplicar, debemos de ir a su html y hacer referencia a una constante que hayamos definido en nuestra hoja de estilos. Por ejemplo, si queremos que debajo de la barra superior de navegación de nuestra página web tengamos una separación del contendo, podemos hacer un estilo main-container, y lo aplicaremos así en nuestro html:

<app-navbar></app-navbar>

<div class="container main-container">

    <router-outlet></router-outlet>

</div>

Y en el contenido del css asociado al elemento, tendremos que especificar lo siguiente:

.main-container {

    margin-top: 20px;

}